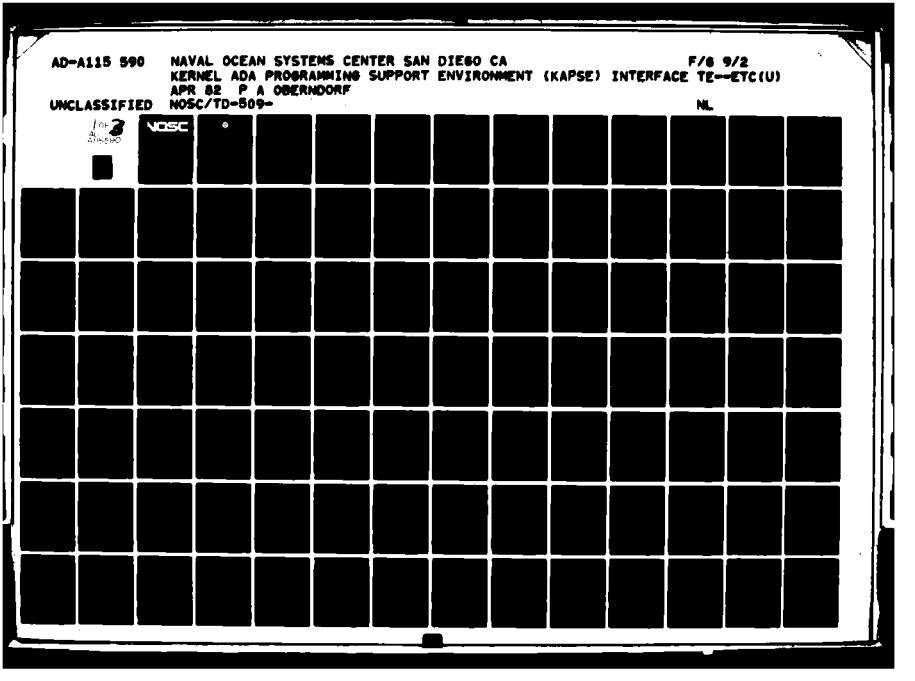
AD-A115 590 NAVAL OCEAN SYSTEMS CENTER SAN DIEGO CA F/G 9/2
KERNEL ADA PROGRAMMING SUPPORT ENVIRONMENT (KAPSE) INTERFACE TE--ETC(U)
APR 82 P A OBERNDORF
UNCLASSIFIED NOSC/TD-509- NL



NOSC

NOSC TD 509

Technical Document 509

KERNEL ADA PROGRAMMING SUPPORT ENVIRONMENT (KAPSE) INTERFACE TEAM: PUBLIC REPORT

VOLUME I

Patricia A. Oberndorf (NOSC),
Chairman

1 April 1982

Prepared for
Ada Joint Program Office
801 N Randolph, Suite 1210
Arlington VA 22209

Approved for public release; distribution unlimited

NAVAL OCEAN SYSTEMS CENTER
SAN DIEGO, CALIFORNIA 92152

NAVAL OCEAN SYSTEMS CENTER, SAN DIEGO, CA 92152

AN ACTIVITY OF THE NAVAL MATERIAL COMMAND

SL GUILLE, CAPT, USN
Commander

HL BLOOD
Technical Director

ADMINISTRATIVE INFORMATION

This report consists primarily of inputs to the KAPSE Interface Team from its joint industry/academia members. The work was sponsored by the Ada Joint Program Office under program element RDAF, project CS22, sponsor order AF0038AJPO-82-2. Although most of the contributions are reproduced here exactly as received, several were retyped to ensure legibility.

Released by
R. A. Wasilausky, Head
C^3I Support Systems
Engineering Division

Under authority of
V. J. Monteleon, Head
Commmand, Control,
Communications and Intelligence
Systems Department

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER NOSC Technical Document 509 (TD 509)	2. GOVT ACCESSION NO. AD-A115 590	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) KERNEL ADA PROGRAMMING SUPPORT ENVIRONMENT (KAPSE) INTERFACE TEAM: PUBLIC REPORT		5. TYPE OF REPORT & PERIOD COVERED 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) KAPSE Interface Team Patricia A. Oberndorf (NOSC), Chairman		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Ocean Systems Center San Diego CA 92152		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS RDAF, CS22, sponsor order AF0038AJPO-82-2
11. CONTROLLING OFFICE NAME AND ADDRESS Ada Joint Program Office 801 N Randolph, Suite 1210 Arlington VA 22209		12. REPORT DATE 1 April 1982 13. NUMBER OF PAGES 232
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Computer language
Ada
Interface standards
Programming support systems

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The initial activities of the Kernel Ada Programming Support Environments (KAPSE) industry-academia interface team are reported. (Ada is a recent, DOD-developed programming language.) The Ada Joint Program Office (AJPO)-sponsored effort will ensure the interoperability and transportability of tools and data bases among different KAPSE implementations. The effort is the result of a Memorandum of Agreement (MOA) among the three services directing the establishment of an evaluation team, chaired by the Navy, to identify and establish KAPSE interface standards. As with previous ADA-related developments, the widest possible participation is being encouraged to create a broad base of experience and acceptance in industry, academia, and the DOD.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-LF-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

CONTENTS

CONTENTS (Continued)

APPENDICES

PREFACE

It is with a great deal of pleasure I present the first of several public reports on the activities of the KAPSE Interface Team (KIT) and its associated industry/academia team. This AJPO-sponsored effort will ensure interoperability and transportability of tools and data bases among different Kernel Ada* Programming Support Environment (KAPSE) implementations. This effort is the result of a Memorandum of Agreement (MOA) among the three services directing the establishment of an evaluation team, chaired by the Navy, to identify and establish KAPSE interface standards. As with previous Ada-related developments, the widest possible participation is being encouraged in order to create a broad base of experience and acceptance in industry, academia and the DoD.

We are at the beginning of a lengthy process that I am confident will lead to a satisfactory strategy for the sharing of APSE tools and databases. I would like to extend my appreciation to those corporations, agencies, governments and individuals who have agreed to make a commitment to this effort. We shall all enjoy the dividends of our investment, both in terms of our shaping and sharing of this advanced technology and of the long range benefits the resulting commonality will bring.

Patricia A. Oberndorf

PATRICIA A. OBERNDORF
Chairman, KAPSE Interface Team

*Ada is a trademark of the Department of Defense (Ada Joint Program Office)

SECTION I

INTRODUCTION

I. INTRODUCTION

This report is the first of several that will be published by the KAPSE Interface Team (KIT). It serves to record the activities that have taken place to date and to submit for public review the products that have resulted. These reports will be issued approximately every six months. They can be viewed as snapshots of the KIT progress, as everything that is ready for public review at a given point in time will be included. By the same token, they also represent evolving ideas, so the contents should not be taken as fixed or final.

Comments on this and all subsequent reports are encouraged. They should be addressed to:

Patricia Oberndorf
Code 8322
Naval Ocean Systems Center
San Diego, CA 92152

or sent via the ARPANET to POBERNDORF@ECLB.

Those who provide feedback will automatically receive the next report.

SECTION II

BACKGROUND

II. BACKGROUND

In December 1980 the Under Secretary of Defense for Research and Engineering established the Ada* Joint Program Office (AJPO) to manage DoD efforts for the introduction, implementation, and life cycle support of Ada. A part of this effort is the coordination of the development of Ada Programming Support Environment (APSE) implementations. The AJPO is responsible for ensuring that DoD has consistent programming support systems which provide the tools needed to develop, manage and support defense system software written in Ada.

In order to coordinate APSE developments, the AJPO obtained a Memorandum of Agreement (MOA) with the three services at the Assistant Secretary level. The tri-service agreement focused on the need to develop a means by which tools and data bases can be readily transported across service-specific APSE implementations. The concept of the KAPSE, as articulated in the APSE STONEMAN document, is the focal point for tri-service commonality. It was agreed that the Army, Air Force, and any other KAPSE efforts within DoD would be closely monitored by the AJPO. The MOA created an evaluation team to be chaired by the Navy and charged with the responsibility of establishing KAPSE interface guidelines, conventions and standards. The MOA concluded by calling for eventual conformance of the current KAPSE efforts to the interface standards established by the KAPSE interface evaluation team. Policy procedures for the implementation of the MOA are now being developed.

Responsibility within the Navy for fulfillment of this MOA rests with the U. S. Naval Material Command (NAVMAT-08Y). NAVMAT has designated the Naval Ocean Systems Center (NOSC) as the lead laboratory for the evaluation and standardization effort. The specified evaluation team has been formed and is called the KAPSE Interface Team (KIT). The team objectives are to define requirements for interoperability and transportability among KAPSEs, followed by guidelines and conventions for achieving them. Finally, these will evolve into standards which, when followed, will ensure the ability of APSEs to share tools and data bases. The membership of the KIT is given in Appendix A of this report.

Initial NOSC tasks included the development of an APSE Interoperability and Transportability (IT) Plan, the designation of team members, and the organization leading to the first KIT meeting. The APSE IT Plan defines the strategy for the development of the interface standards, team composition, and a schedule of milestones and deliverables.

The AJPO responsibilities are formally for the management of Ada within DoD; however, to achieve this goal the AJPO must be sensitive to the world software community. To date the Ada program has found success in an open style of public disclosure and sensitivity to public feedback. In view of this success, it was decided to supplement the KIT with a team of representatives from industry and academia. Drawing on the industry/academia participants in the Ada language effort, a solicitation was made for APSE IT participation. The response was impressive in view of the fact that all involvement would have to be funded by the participants. The motivation for supplementing the DoD oriented KIT with an

*Ada is a trademark of the Department of Defense (Ada Joint Program Office)

industry/academia team was to provide the KIT with a broad base of inputs, reviews and advice from the technically qualified talent in industry and academia. The membership of the industry/academia team is given in Appendix B of this report.

The KIT held its first meeting in San Diego, California, on January 19 and 20, 1982. This initial meeting served to develop organizational relationships and to define an approach consistent with the APSE IT Plan. Working groups were formed, task assignments made, and future meeting schedules established. A critical objective of this first meeting was the definition of APSE generic terms and the formulation of KAPSE interface categories. The results are contained in Appendices C, D and E of this report.

The first meeting of the industry/academia team was held from 17 February through 19 February, 1982. The meeting served as a forum to define the role of the group in support of the KIT, establish organizational relationships, and define working groups to parallel the KIT effort. Each member of the team presented a point paper to express salient concerns affecting APSE IT. The point papers constitute Section IV of this report.

Future meetings of the teams are currently scheduled every two or three months to allow the working groups time for analysis and preparation and to ensure coordination of the APSE IT efforts. There will be a continuing effort to provide public availability of appropriate point papers and status documents prepared by the KIT. This report is in step with the policy of maintaining public visibility of Ada-related activities.

SECTION III

MEETING PROCEEDINGS

A. KAPSE Interface Team (KIT)

B. Industry/Academia Team

KAPSE Interface Team (KIT)
Meeting of 19, 20 January 1982
San Diego, California

Time	Proceedings

Tuesday 19 January

0840 Tricia Oberndorf (NOSC)
Brings the meeting to order.

0845 Keynote Presentation - Jack Kramer
AJPO, Navy and the KIT

0900 Introduction and Organization -
Tricia Oberndorf

- o APSE IT Plan will be distributed to KIT members when available
- o Expect the KIT to meet 4 times a year - twice a year in San Diego
- o IAT will meet 3 or 4 times a year

1000 BREAK

1015 Definition Session - Team effort in defining the following words:

- o Interoperability
- o Transportability
- o Reusability
- o Host
- o Target
- o Rehostability
- o Retargetability

1200 LUNCH

1250 Definition Session (continued)

1430 BREAK

1445 KAPSE Interface Categories Session

Session Objectives - Tricia Oberndorf

General discussion

Initial suggestions for KAPSE Interface Categories are developed, along with a description of each category.

- o Data Base Services
- o Configuration Management
- o Device Interaction

Time	Proceedings

- o Run-Time System
- o Command Language Interfaces
- o Special - Character/Control-Key Dependencies
- o Suspension/Interruption/Continuation Mechanisms
- o LOGON/LOGOFF Services
- o Inter-Tool Invocation Mechanisms
- o Ada Intermediate Language

1700 Adjourn for the day

Wednesday 20 January 1982

0830

0845 KAPSE Interface Categories Session (continued)

- o Other Compiler Interfaces
- o Performance Measurement Capabilities
- o Recovery Mechanisms
- o Other Operating System Services
- o Other Tool Interactions
- o *Support for Targets*

1030 BREAK

1040 Discussion of New Categories and Assignment to Work Groups

Group		
I.	KAPSE User Support o Device Interactions o Program Invocation and Control o LOGON/LOGOFF Services	Group Leader: Doug Johnson Members: Jack Kramer Larry Johnston
II.	Data Interfaces o Data Base Services o Ada Intermediate Language o Inter-Tool Data Interfaces	Group Leader: Donn Milton Members: Tricia Oberndorf Bob Bartholow
III.	KAPSE Service Interfaces o Ada Program Run-Time System o Performance Measurement Capabilities o Bindings and Their Effects on Tools	Group Leader: John Oldham Members: Warren Loper Ron House

Time	Proceedings	
	IV. Miscellaneous & 'Non-Categories'	Group Leader: Hal Hart
	o Recovery Mechanisms o Support for Targets o Distributed APSEs o Security	Members: Guy Taylor, Dan Sterne
1100	Working Groups Session Each group breaks up and discusses their assigned categories.	
1200	LUNCH	
1315	Each Working Group reports on their morning assignments.	
1450	A summation of the groups' efforts is presented by Hal Hart (TRW).	
1600	A synopsis of the Navy's position on APSE development is presented by LCDR Jack Kramer.	
1630	END OF MEETING	

Industry/Academia Team
Meeting of 17, 18, 19 February 1982
San Diego, California

Time	Proceedings
Wednesday 17 February	
0815	Tricia Oberndorf (NOSC) brings the meeting to order.
0820	Keynote presentation - LCDR Jack Kramer of the AJPO.
	o The role of the Industry/Academia Team
	o The Navy's concerns on KAPSE Standardization.
	o The AJPO objectives for the APSE IT effort.
0845	Introduction and Organization - Tricia Oberndorf (NOSC).
	o History of events leading to the APSE IT effort.
	o Composition of the KAPSE Interface Team, the KIT executive committee, and supporting contractors.
	o Scope of the APSE IT effort.
	o Organizational relationships of participating members.
	o Related efforts.
0855	A question and answer period followed allowing attendees to clarify the role and goals of the industry/academia team.
0915	BREAK
0930	Reconvened by Tricia Oberndorf (NOSC). Introduction to presentation of point papers.
0940	1st Paper Dennis Cornhill, Honeywell

0955	2nd Paper	Fred Cox Georgia Institute of Technology
1015	3rd Paper	Jon Fellows System Development Corporation
1030	4th Paper	Herman Fischer Litton Data Systems
1050	5th Paper	Roy Freedman Hazeltine Corporation
1105	6th Paper	Anthony Gargaro Computer Science Corporation
1125	7th Paper	Steve Glaseman Teledyne Systems
1145	8th Paper	Eric Griesheimer McDonnel Douglas Astronautics
1200	LUNCH	
1315	General adminsitrative remarks leading to continuation of point papers	
1325	9th Paper	Ron Johnson Boeing Aerospace Corporation
1340	10th Paper	Judy Kerner Norden Systems
1355	11th Paper	Reed Kotler Lockheed Missle and Space
1410	12th Paper	Eli Lamb Bell Labs
1425	13th Paper	Tim Lindquist Virginia Institute of Technology
1445	14th Paper	Doug Locke IBM
1500	15th Paper	Tim Lyons Software Science Ltd (United Kingdom)
1520	BREAK	
1530	16th Paper	David McGonagle General Electric

1545	17th Paper	Charles Mooney Grumman Aerospace
1600	18th Paper	Ann Reedy Planning Research Corporation
1620	19th Paper	Sabina Saib General Research Corporation
1640	20th Paper	Edgar Sibley Alpha Omega Group, Inc.
1655	21st Paper	Herb Willman Raytheon
1705	Adjourn for the day.	

Thursday 18 February

0800	Reconvened and proceeded with point papers.	
0810	22nd Paper	Jim Ruby Hughes Aircraft
0820	23rd Paper	Rob Westermann TNO-IBBC (The Netherlands)
0835	24th Paper	Doug Wrege Control Data Corporation
0855	25th Paper	Larry Yelowitz Ford Aerospace
0910	26th Paper	Harrison Morse Frey Federal Systems
0935	27th Paper	Thomas Standish University of California at Irvine
1005	BREAK	

1020 Definition Session. A group effort at smoothing definitions prepared by the KIT. Definitions addressed include:

- o Interoperability
- o Transportability
- o Reusability
- o Host
- o Target
- o Rehostability
- o Retargetability

1115 KAPSE Interface Categories Session

KAPSE Interface Category Worksheets prepared by the KIT are elaborated upon for content, issues presented, and required actions.

- o Program Invocation and Control
- o LOGON/LOGOFF Service
- o Device Intractions
- o Data Base Services
- o Inter Tool Data Interfaces
- o Ada Intermediate Languages
- o Ada Program Run-Time System (RTS)
- o Binding and their effect on tools
- o Performance measurement capabilities
- o Recovery Mechanisms
- o Distributed APSEs
- o Security
- o Support for Targets
- o Pragma and other tool controls

1210 LUNCH

1330 Reconvened. An election was held for the industry/academia team organization. Election results were:

Chairman:	Edgar Sibley	Alpha Omega Group
Secretary:	John Oldham	TRW

1355 Discussion followed to allow further clarification of the team goals, relationship to the overall APSE IT effort, and definition of potential end products from the team effort.

1535 BREAK

1550 Organizational Session. Open discussion of alternate approaches to development of the APSE IT requirements. Aspects considered included development of strawman interface categories, and DoD management options. A subset of the options addressed included:

- o DoD approval of independently developed APSEs.
- o DoD development and control of a single KAPSE.
- o Definition of an ANSI Standard.
- o Define a defacto standard such as UNIX.

1645 The team concludes that the best approach is to parallel KIT efforts. Working group assignments are made in consonance with the KIT allocation of interface categories.

Group I KAPSE User Support	Group Leader:
Device Interactions	Harrison Morse
Program Invocation and Control	Members:
LOGON/LOGOFF Services	Anthony Gargaro
	Charles Mooney
	Tim Lindquist
	Fred Cox
Group II Data Interfaces	Group Leader:
Data Base Service	Judy Kerner
Ada Intermediate Languages	Members:
Inter Tool Data Interfaces	Herman Fischer
	Ann Reedy
	Edgar Sibley
	Tim Lyons
	David McGonagle
	Roy Freedman
Group III KAPSE Services	Group Leader:
Ada Program Run-Time System (RTS)	Sabina Saib
Performance Measurement Capabilities	Members:
Bindings and their effects on tools	Tim Standish
	Eli Lamb
	Herb Willman
	Doug Wrege
	Ron Johnson
	Jon Fellows
	Doug Locke
Group IV Miscellaneous	Group Leader:
Recovery Mechanisms	Steve Glaseman
Support for Targets	Members:
Distributed APSEs	Eric Griesheimer
Security	Larry Yelowitz
	Rob Westermann
	Dennis Cornhill
	Jim Ruby

1700 Adjourn for the day.

Friday 19 February

0815 Reconvened. General Announcments.

0830 Working Group Session.
The team breaks into the working groups to discuss assigned KAPSE Interface Categories.

1020 Working Groups adjourn and the team at large is reformed. A general discussion on future meetings develops the following industry/ academia team schedule:

Date	Place
April 28,29,30	Washington, D.C.
June 18,19,20 (Tentative)	Boston
October 4,5,6,7	Washington, D.C.

1115 Each working group reports on morning activities. The individual groups will be developing a charter, establishing lines of communication, elaborating on the IT generic definitions, and begin evolving requirements applicable to the specific area of assignment.

1140 A summation of events and presentation of subjects for possible elaboration by the team at large - Tricia Oberndorf (NOSC).

1200 END OF MEETING

SECTION IV

POINT PAPERS

A.	Dennis Cornhill	Honeywell
B.	Fred Cox	Georgia Institute of Technology
C.	Jon Fellows	Systems Development Corporation
D.	Herman Fischer	Litton Data Systems
E.	Roy Freedman	Hazeltine Corporation
F.	Anthony Gargaro	Computer Science Corporation
G.	Steve Glaseman	Teledyne Systems
H.	Eric Griesheimer	McDonnel Douglas
I.	Judy Kerner	Norden Systems
J.	Reed Kotler	Lockheed Missles and Space
K.	Eli Lamb	Bell Laboratory
L.	Tim Lindquist	Virginia Institute of Technology
M.	Doug Locke	IBM
N.	Tim Lyons	Software Science Ltd (United Kingdom)
O.	Charles Mooney	Grumman Aerospace
P.	Ann Reedy	Planning Research Corporation
Q.	Sabina Saib	General Research Corporation
R.	Edgar Sibley	Alpha Omega Group, Inc.
S.	Herb Willman	Raytheon
T.	Jim Ruby	Hughes Aircraft
U.	Rob Westermann	TNO-IBBC (The Netherlands)
V.	Doug Wrege	Control Data Corporation
W.	Harrison Morse	Frey Federal Systems
X.	Thomas Standish	University of California at Irvine
Y.	Pekka Lahtinen	Oy Softplan Ab (Finland)
Z.	Stuart Schaffner	Massachusetts Computer Associates

HOST ARCHITECTURE AS A KAPSE INTERFACE ISSUE

Dennis Comhill
(Honeywell)

Introduction

Stoneman carefully emphasizes the machine independentness of the KAPSE interface. Paragraph 1.E which describes the APSE levels, states that the KAPSE "... presents a machine-independent portability interface." The same idea is repeated in paragraph 2.B.11 where the KAPSE is characterized as a "virtual support environment (or a 'virtual machine') for Ada programs ..." Paragraph 3.H states that "an APSE shall be portable so far as practicable" and that this goal will be achieved by following the KAPSE design model defined by Stoneman. In addition to these specific requirements for machine independence, the KAPSE requirements have been carefully written to minimize assumptions about the base on top of which the KAPSE is built.

Clearly the intent of the KAPSE is for it to present a machine independent interface to the MAPSE. But this implies that KAPSE standards must not favor (or disfavor) a particular machine or system architecture.

Influence of Initial APSE Implementations

The construction of an APSE is complicated. We tend to approach early attempts at complicated projects by trading off complexities which are not inherent to the problem, against other parameters such as cost, performance or convenience of use. For the APSE, one way to minimize project risk is to select a host which is mature, well understood, has high performance and which contains a powerful and flexible operating system. As a result, the initial hosts are likely to be architecturally similar because their architectures reduce project risk in the early 1980's. But they could very well not have the most cost effective architecture for the long term.

These early implementations are certain to influence a KAPSE standard. The danger is that their influence could result in a standard which discriminates against architectures which are more suitable.

Is Architecture an Issue?

Traditionally, software development of large programs has been hosted on centralized computers. Is this likely to change?

Hard data is lacking but changes in the economics of software development at least suggest that centralized computers may not be the long-term solution for the APSE host. Specifically, hardware costs have decreased while programmer costs have increased. This has been the trend over the past 30 years and should continue for the indefinite future.

The Steelman requirements recognize this economic trend and so Ada has been designed to shift more of the development burden off the programmer and onto the compiler than is the case for the languages it is intended to replace. But this now means that the host must be capable of economically delivering the large quantities of central processor and memory resources needed by the compiler. Otherwise, the programmer will be forced into working around the language by minimizing the number of compilers, working during off-shifts and defining compilation units on the basis of compilation efficiency rather than readability.

A second consequence of the trend in ecomonics has been the emergence of microprocessor based software development work stations. Experience with them has shown that a station containing a 16/32 bit microprocessor with 256K bytes main memory, a 10M byte Winchester disk, a floppy disk and CRT (currently available for $10K - $20K) can support an extremely cost effective software development environment for a single user.

The evidence does not prove that microprocessor based systems are more cost effective APSE hosts than large computers, but there are some hints that this may be the case.

The Ada Test Translator is a Multics hosted tool for compiling and interpreting programs written in preliminary Ada. It compiles at a rate 250 times slower than Multics' pl1 compiler*. Though well designed, the Test Translator was not intended to compile efficiently, and in any case we should not expect Ada compilers to be as fast as pl1 compilers. Nevertheless it is by no means certain that the performance of the Test Translator could ever be elevated to an acceptable level on Multics.

---

*Benchmark tests were performed by the Test Translator development team.

The Ada/Ed interpreter developed by NYU is hosted on a VAX. It too was not intended to be efficient and as a consequence its performance is also extremely poor. However, it appears that NYU's VAX configuration can support only four concurrent compilations. With six concurrent compilations, their system degrades to the extent that little progress is made by any of the compilations.* Can Ada/Ed's efficiency be raised to an acceptable level, or is it that the VAX is not a cost effective Ada host?

TeleSoft has been selling a subset compiler hosted on Motorola 68000 based work stations. For large programs (which minimize the effect of paging the compiler into main memory) they claim a remarkable compilation speed of 500 lines per minute. This figure should be partially discounted since their compiler is a subset implementation. Still it does provide evidence that microprocessors may be able to deliver APSE computing resources efficiently.

This data is not offered as proof that microprocessor work stations are better hosts than large computers. In fact, a single user work station is an entirely inappropriate APSE host. However, the data does suggest that a network of single user work stations might form a good host. Therefore it is important that KAPSE interface standards not preclude a network architecture, at least.

KAPSE Requirements

Stoneman places relatively few requirements on the KAPSE. Sections 5.A and 5.C list those functions which must be supplied by the KAPSE for the MAPSE: database facilities, runtime support for Ada programs, I/O, tool invocation and asynchronous user control from an interactive terminal. Section 5.E continues by listing some abstract data types which must be defined by the KAPSE in order to facilitate intertool communication.

The requirements are not specific and so allow for a great deal of interpretation. For example, I/O is typically implemented in layers where the more primitive levels access the hardware directly and build up to a user interface layer. For the APSE, the user layer interface is defined by the Ada Language. But many of the lower levels also meet the Stoneman requirements and so could

---

*Conversation with Gerry Fisher of the Ada/Ed project.

qualify for the KAPSE I/O interface. However, the low levels of I/O tend to be machine specific. For example, they might imply close coupling to the peripheral which is not the general case for all potentially useful hosts.

Selecting the correct level of functionality at the KAPSE interface also impacts how well a particular architecture can be used. Inside the KAPSE interface are the KAPSE implementation and host operating system and hardware - a mixture of both hardware and software. Outside the KAPSE interface are the MAPSE and tool extensions - all software. The KAPSE interface defines the extent of the useful functionality of the hardware. Functions implemented in hardware at a higher level than the KAPSE interface are wasted because the MAPSE cannot use them; it is locked into a contract with the KAPSE interface.

Consider I/O as an example. It could be implemented at the KAPSE interface level with GETs and PUTs for the intrinsic types INTEGER, FLOAT, CHARACTER, etc. Therefore a GET at the Ada program level, for a record type consisting of four components, two integers, a floating point number and a character, would be translated into four separate GETs by the code generator and runtime. But this would be inefficient if the peripheral could deal with the entire record at once so that it could process the GET with a single invocation.

Conclusions

We don't yet know what the most cost effective APSE host architecture is, if indeed only one exists for all applications. Therefore KAPSE interface standards should be carefully expressed to be machine independent.

Initial APSE implementations are for architecturally similar computers - large, centralized mainframes. While experience from these products will certainly be useful in defining standards, this experience should not be allowed to compromise the machine independentness of the standards.

Striving for machine independentness is likely to drive the KAPSE interface into higher levels of abstraction, which is likely to also make the KAPSE larger. Thus there is a conflict between the goals of having a small KAPSE and a machine independent KAPSE interface.

Although this paper only considers a network of microprocessors as an alternative to centralized hosts, clearly other architectures should also be considered when validating the machine independentness of the standard.

Stoneman describes the KAPSE as a "virtual machine for Ada programs." This theme should be an important factor in defining KAPSE interface standards.

PORTABLE WHAT?

Fred Cox
(Georgia Institute of Technology)

Abstract

Currently, the Department of Defense is funding the development of two different Ada Programming Support Environments (APSEs). If both APSEs continue to develop in their current directions, much of the benefit expected from standardizing on Ada as a programming language may be lost. Software contractors may have to invest in two different machines in order to develop software for both the Air Force and the Army. Because of differences between the APSEs in the area of the user interface (especially in the command languages and data base structures), transportability of development support software and application systems may be low. Programmer productivity may be degraded by problems in moving from APSE to APSE, machine to machine, command language to command language, and toolset to toolset. Maintenance personnel may consequently have to contend with a proliferation of user interfaces produced by development teams in order to hide differences between the APSEs. Therefore, it is important that both the machine and the user interfaces be standardized.

Context

In 1975, the Department of Defense initiated the Common High Order Language program with the objectives of lowering the costs and enhancing the quality of defense software. The product of this effort is the Ada programming language. The design of Ada was highly influenced by the facts that 1) embedded computer systems account for more than half of DoD's software costs (Fisher 1) and 2) 70% to 90% of software life cycle costs are incurred during the maintenance and support phase (Stoneman 2.A.1 2). Currently, the Department of Defense is funding

the development of Ada Programming Support Environments (APSEs) in recognition of the need for such support if the objectives of lower costs and higher quality are to be fully realized with Ada.

The requirements for APSEs are set forth in the Stoneman document 2 . Paragraphs 1.B and 1.C of that document reveal the purpose and principal features of an APSE:

> 1.B The purpose of an APSE is to support the development and maintenance of Ada applications software throughout its life cycle, with particular emphasis on software for embedded computer applications.
>
> 1.C The three principal features of an APSE are the data base, the (user and system) interface and the toolset. The data base acts as the central repository for information associated with each project throughout the project life cycle. The interface includes the control language which presents an interface to the user as well as system interfaces to the data base and toolset. The toolset includes tools for program development, maintenance and configuration control supported by an APSE.

In the Stoneman, a particular approach to the design of APSEs is sug gested, whereby an APSE would be based on a minimal set of tools and made portable by an host-independent interface to a kernel of support functions. The toolset would be called the MAPSE and the kernel of support functions the KAPSE. The primary reason for this approach is to provide for portability (Stoneman 2.B.10). The KAPSE in particular is responsible for ensuring this portability. Again quoting from the Stoneman:

> 2.B.11 The purpose of the KAPSE is to allow portable tools to be produced and to support a basic machine-independent user interface to an APSE. Essentially, the KAPSE is a virtual support environment (or a "virtual machine") for Ada programs, including tools written in Ada.
>
> 2.C.4 ... A further intent of the Stoneman, however, is to propose a way forward towards the goal of portability. APSEs will address this requirement where relevant by making explicit a KAPSE portability level.

2.C.5 In order to achieve the long-term goal of portability of software tools and application systems dependent on them, it is intended that conventions and, eventually, standards be developed at the KAPSE interface level. This level will then serve as specifying a portability interface and separate tools or integrated sets of tools which meet the KAPSE interface requirements will be readily portable between hosts.

Some questions might be raised at this point. What is it that is to be transported, and what in particular is required to ensure its portability?

Portability

As to what is to be transported, a number of candidates present themselves. Perhaps the most obvious need is for APSEs to be portable from host machine to host machine. Not only may MAPSEs need to be used on different host machines, they may need to be compatible with different APSEs. Individual Ada programs (in particular tools) may need to run on different machines and with different APSEs and MAPSEs. Application systems (along with their immediate support systems) may be shifted not only among host machines, APSEs and MAPSEs, but among development groups and from development groups to maintenance and support groups. In addition, software personnel themselves will have to use different host machines, APSEs and MAPSEs and will have to move from project to project and application system to application system, often having to work on more than one concurrently. If the KAPSE interface is responsible for ensuring portability, it must address the portability requirements of each of these areas. What then are these requirements?

In general, the APSE depends on common operating system functions provided by the host. Such functions include the logging-in and logging-off of users, user or process control of executing programs (e.g., initiation, suspension, resumption and termination), protection of users, processes and data objects, run-time support for executing Ada programs, data storage and retrieval, and communication facilities. Included in communication functions are both inter-system and intra-system communi-

cations, such as those between users, processes (e.g., tools), processes and I/O devices, and users and processes. These functions are provided through what the Stoneman calls the system interface (Stoneman 1.C).

Some of the portability requirements for MAPSE tools are called out in paragraphs 5.E.3 through 5.E.9 of the Stoneman and include inter-tool data structures such as a definition for the source representation for a compilation unit (for the translator, prettyprinters and sytax-directed editors), the abstract syntax definition of a compilation unit (as above), the translator's intermediate language representation of a compilation unit (for code generators), the abstract data type definition of an executing Ada program (for loaders and debuggers), the definition of a compilation unit's symbol table (for debuggers and cross reference analysis) and the definition of the program library (for library management and configuration control). (The above lists of example tools are not to be considered complete.)

Many individual tools, especially those supporting application systems, will be dependent on data base conventions, I/O conventions and the structure of the command language. Perhaps the most prevalent form of tool will be the command procedure. Command procedures are used for a variety of purposes such as providing abbreviated forms of frequently used system calls and sequences of system calls, establishing links between processes (e.g., pipelining) and hiding problems in the command language or system such as deficiencies, complexities or verbosity. Examples are invoking the translator with certain options, and calling the editor for input, formatting the output, returning to the editor for review, deleting the working files, and optionally printing the resulting objects, iterating or exiting. Command procedures can be very beneficial if they reduce the complexity and effort for development and maintenance personnel, thereby enhancing their productivity. Unfortunately, command procedures often are needed to insulate the users from a poorly engineered human interface.

Both command procedures and software personnel are highly dependent on the command language for invoking APSE functions. They and the application systems dependent on them are also dependent on the structure of and methods of interacting with the data base. Consequently,

serious difficulties are foreseen in trying to move personnel, command procedures and project support from APSE to APSE if the command language and data base interfaces are not standardized. These difficulties might seriously degrade the productivity of development and maintenance personnel, since they would have to contend with a variety of command languages and data base structures. Under these circumstances, development teams might begin to produce their own interfaces (e.g., customized command language interpreters) in order to hide the differences in the APSEs used. This move would compound the variety of interfaces that maintenance personnel would have to work with. These problems run counter to the spirit of the guidelines of uniform protocol and project portability set forth in Stoneman, paragraphs 3.G and 3.J.

At Present

The Department of Defense is now funding the development of two very different APSEs, the Ada Language System (ALS) and the Ada Integrated Environment (AIE). Other APSEs are expected to be produced by industry and the academic community. From information appearing in the design specifications for the ALS as presented for review in 1981, it appears that the ALS will have problems in rehosting. Much of the difficulty seems to stem from an inadequate specification of a logical KAPSE interface. Virtual memory seems to be required of a host. File handling, inter-tool communication, error severity reporting and security seem to be VAX-11/780 dependent. Certain features of Ada (such as representation specification) have even been restricted, apparently in deference to the VAX. Considerations for rehosting seem not to have been pursued much beyond those necessary for rehosting the ALS on a different operating system, UNIX/32V, but still on the same machine.

If both the ALS and the AIE are completed and required by their respective branches of the military for use by their software contractors, much of the cost reduction anticipated through the use of Ada may be lost. Development personnel will have to contend with different command languages and data base structures with the resulting problems discussed above. Additionally, contractors doing business with both the Air Force and the Army may have to acquire and support two different

development machines, an IBM 370 or Interdata 8/32 for the AIE and a VAX-11/780 for the ALS. It is also likely that much development support software and project management software will not be portable and therefore not shareable across systems, thereby requiring duplication of effort in their development and maintenance.

Discussion

The KAPSE may be viewed in a number of different ways. It might be considered as that part of the APSE that provides for host-dependent services. It might be expected to provide not only those services, but a high order data base facility as well. The view espoused here is that the KAPSE should be defined as that part of the APSE that provides for portability in general, as opposed to providing for only the machine portability level. This would require that the system, data base and user interfaces all be part of the KAPSE interface specification. This approach seems consistent with that of the AIE Statement of Work 3 . Although the philosophy of portability indicated in the Stoneman seems to support this approach, the implementation suggested by the Stoneman seems to lean toward the more limited views mentioned. At issue, then, are the questions of what degree of portability is desired, what requirements are entailed by that degree of portability, and how shall those requirements be met so that portability is ensured.

Summary

The development of the APSE, and therefore of the KAPSE, is motivated to a large degree by a desire to reduce the cost of developing and maintaining defense software. Much of this cost reduction is predicated on sharing software and enhancing the productivity of software personnel. These objectives require the portability of both software and personnel. The KAPSE's purpose is to ensure portability. Not only must APSEs and MAPSEs be portable from machine to machine, individual tools, application systems and software personnel must be able to move from APSE to APSE as well as from machine to machine. To achieve these levels of

portability, standardization is necessary for the system, data base and user interfaces. Therefore, each of these interfaces should be specified in the KAPSE interface specification.

References

1. Fisher, David A., DOD's Common Programming Language Effort, "Computer", March 1978, pp. 25-33.

2. Buxton, John N., Requirements for Ada Programming Support Environments - "Stoneman", Department of Defense, February 1980.

3. Revised Statement of Work for Ada Integrated Environment, PR No. B-0-3233, Rome Air Development Center, Griffiss Air Force Base, New York, 26 March 1980.

IMPLEMENTING THE ADA PROGRAMMING SUPPORT ENVIRONMENT IN ADA

Jon Fellows
(Systems Development Corporation)

Abstract

This paper examines the issues involved in using Ada to implement its own environment. Recent experience with the languages Concurrent Pascal, Modula, and Mesa demonstrates the practicality of implementing operating systems and other system software in high level languages. In particular, Wirth's Lilith project demonstrates that a complete programming support environment can be implemented with no use of lower level languages.

We begin with the view of an executable Ada program as a collection of modules, some of which were programmed by application programmers and some of which were programmed by system programmers. Different modules are bound together at different times and for differing durations. The dependencies among modules can be represented by the graph of the "uses" hierarchy. This graph is simply an extension of the graph of the partial ordering of a programs' compilation units. The extensions incorporate those Ada components which implement system primitives and Ada run time support.

We propose an implementation of this concept in a single user system similar to the Solo system written in Concurrent Pascal or the Lilith system written in Modula. We then discuss the performance issues of this approach and its extension to multiple user environments.

Introduction

This paper examines the issues involved in using Ada [1] to implement its own environment. Recent experience with the languages Concurrent Pascal [2], Modula [3], and Mesa [4] demonstrates the practicality of implementing operating systems and other system software in high level languages. In

particular, Wirth's Lilith [5] project demonstrates that a complete programming support environment can be implemented with no use of lower level languages. Earlier systems by Brinch Hansen (Solo) [6] and Xerox PARC (Pilot) [7] also provided convincing demonstrations of the feasibility of this approach.

We prefer high level modular languages for system programming for the same reasons that these languages are widely preferred for applications programming, including ease of maintenance, enhanced productivity, and their contribution to elegance of design. Computer architectures which directly support the concepts of such languages enhance the performance of this approach.

Throughout this paper, the terms component, module, and compilation unit will be used interchangeably.

The Ada Environment

For the purposes of this paper, the Ada environment consists of a set of services which support the management, design, implementation, and execution of Ada programs. These services will take forms that include: informal methods such as management procedures and directives; formal methods supported by Ada programs and the data bases they manage; and low-level execution support provided by predefined Ada tasks and packages which can be bound to user programs.

This paper addresses the following four classes of Ada environmental services and proposes their implementation almost entirely in Ada:

1. System Services: Input/Output, Directory operations, Inter-program calls, etc.

2. Run Time Support: Process Scheduling, Memory Allocation, etc.

3. KAPSE Tool Set: Basic program development tools

4. APSE Tool Set: Full Software Engineering Environment

The implementation in Ada of the KAPSE and APSE [8] tool sets should not be controversial at this time. However, few (if any) of the current KAPSE efforts propose the implementation of system services and run time support in Ada. We feel that this represents a missed opportunity.

The Vanishing Operating System

A traditional monolithic operating system must attempt to support all possible uses of the system that it manages. As a result, the system must be quite general, and therefore complex. By generality, we mean an attempt to be all things to all users. Such systems typically consume a large portion of the physical resources of a system. In addition, the necessarily general mechanisms presented at a user interface are frequently much less efficient than mechanisms more suited to a specific user's application.

Perhaps the most significant drawback of this existing approach is that the complexity of the resulting systems challenges our current ability to produce reliable systems. Even if a monolithic system converges through use and maintenance to a reasonably reliable system, the addition of new features frequently re-introduces the reliablility problem all over again.

The capability for separately compiling and binding software components, present in Ada and several of its predecessors, provides the opportunity to view an operating system as a collection of predefined components. Only the needed components then need to be bound to a user program in order for it to be executed. Users that desire a general interface can use very high level components, while users who require precise control of their environment can bypass these and use lower level components. The library management component can enforce controls over this privilege. There is no need to view operating system components as different than other predefined (re-usable) components in a system library.

The relationships, or access rights, of the components of a program can be depicted as the graph of the "uses" relation. In Ada terms, this means that the "uses" relation is captured as the graph of the partial ordering imposed by the "with clauses" of the Ada program. We are simply advocating that this relationship extend down to the level of the bare machine.

Figure 1 gives an example of such a structure of software components, and identifies system components, the user program interface, and the user program's components.

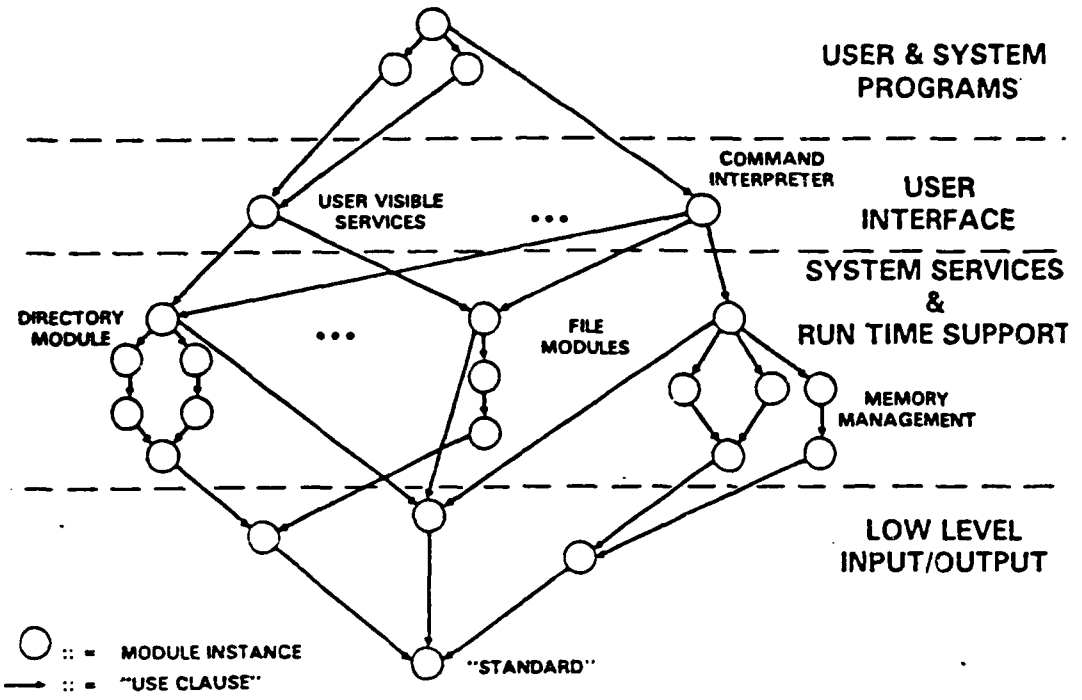


Figure 1: A typical system access graph (incomplete)

Implementation Strategy

We begin with the view of an executable Ada program as a collection of modules, some of which were programmed by application programmers and some of which were programmed by system programmers. Different modules are bound together at different times and for differing durations. The dependencies among modules can be represented by an extended graph of the "uses" hierarchy. The extensions incorporate those Ada components which implement system primitives and Ada run time support.

There are three key issues that must be addressed by this approach: defining the access rights of components, binding instances of components, and efficiency. The following subsections address these issues.

Access Rights

Access rights between components are declared using the library facilities of Ada. The "with clauses" of modules define components that can be named inside the module. Clearly, the access of a programmer defined component to other programmer defined components and to predefined components providing system services is declared explicitly in the text of the component being defined. It is a relatively simple matter for the compiler to define these access rights in the Ada library.

Ada run time support poses a special problem, since it implements features of the language which are difficult to express in the language itself. It would be difficult, for example, to specify task rendezvous as a set of packages and still retain current Ada syntax. We propose that the dependence of an Ada program on its run time mechanisms should not be explicitly declared in the program; this dependence must be detected by a compiler which is aware of the existence of these services and defines the access of a compilation unit to these services. Although such access is not visible in the program text, it must be described in the Ada library and any reports the library generates.

Module Binding

Once the access relations of the components of an Ada program have been defined in the library, the problem then becomes that of binding the components together into an executable unit. In the following we will refer to the actions of a linker, which statically binds components to each other; and a loader, which may dynamically bind components that are already being used.

In the simplest case, that of a single user system that executes one Ada program at a time, we can envisage the execution of Ada programs by the consecutive loading of executable images into the computer's store. Each program is bound in its entirety by the linker; and the loader loads everything required for the execution of the program, regardless of what has been used by the previous program. The command interpreter program would be loaded in between invocations of user programs and would cause the loading of user identified programs.

A more advanced (and more useful) system would permit concurrent execution of Ada programs in order to serve more than one user or to provide background execution capabilities. In such a system, the executable code (and possibly data) of a component might be described as a segment in the store. We could then specify the joint use of a system component by more than one user program. This capability implies that the loader has dynamic binding capabilities. Such a capability is essential for the sharing of modules which retain state information, such as resource monitors.

The design of the interactions between the loader and the library is clearly a key design issue for this approach. This design should be approached carefully, with well defined increments of capability.

Efficiency

This problem must be addressed in any case for embedded computer systems. Early experience in solving the efficiency problems of Ada (unpleasant to be sure) is highly desirable. In extreme cases, it is possible that low level modules will end up being implemented in assembler or microcode. Hopefully this would be done after performance evaluation of the Ada system. In any case, the introduction of architectures optimized for the execution of Ada programs will help alleviate this important problem.

Incremental Development

In order to get a grasp on the basic issues of this approach, we propose an implementation of this concept in a single user system similar to the Solo system written in Concurrent Pascal or the Lilith system written in Modula. Once these issues have been mastered, we could proceed to add functions in reasonably sized increments. Table 1 presents such a staged development plan:

INCREMENT	FEATURES
Sequential Single User	Substantial System Services, Simple File Directory, Simple Run Time Support Simple Binder (no latency) APSE Program Development Tools
Concurrent Single User	All of the above features, Multi-program binding, Pipelines, Input/Output Spooling, Background program execution
Concurrent Multiple User	All of the above features, Multi-level File system, Shared Resource Management, Security Monitor, APSE Program Management Tools

TABLE 1: INCREMENTAL DEVELOPMENT PLAN

Conclusion

We have presented an immodest proposal for the development of the entire Ada program support environment in Ada itself. The development of entire systems in high level languages has been adequately demonstrated for small systems. We can now attempt to demonstrate that this approach works for a large application such as the Ada environment.

In many ways, this proposal suggests nothing more than taking our own medicine; i.e. demonstrating the utility of Ada concepts in addressing embedded computer system issues.

We strongly feel that this approach brings many benefits to the Ada community. These benefits are briefly reviewed below.

Conceptual Compatibility

It is highly desirable that the KAPSE interface to user programs present a set of services that are consistent with the language concepts of Ada. ıch consistency is absent from some currently proposed KAPSEs. What better way to achieve this goal than to implement these services in Ada?

Early Experience in System Programming in Ada

System programming shares concerns with many embedded system applications. Among these concerns are the synchronization of concurrent processes, the management of shared resources, interfacing to peripheral devices, and establishing shared data bases. Implementing the Ada environment in Ada gives early experience with these important embedded computer system issues.

Re-usable Components

Certain of the tasks and packages from the Ada environment would be expected to be useful as initial elements of a library of re-usable components for embedded computer systems. This is one of the key pay-off areas for an Ada environment, and again, early experience is desirable.

APSE Portability

One of the major issues for an Ada library management system is the definition and control of versions of programs. As an example of dealing with this problem, the Ada environment can be made portable by implementing different versions of the KAPSE

that use different versions of low level components. These low level components should encapsulate the machine specific details of the KAPSE.

References

[1]. MIL_STD 1815, "Military Standard, Ada Programming Language" United States Department of Defense, December 1980.

[2]. Brinch Hansen, P. "The Programming Language Concurrent Pascal," IEEE Transactions on Software Engineering 1, 2 June 1975.

[3]. Wirth, N. "Modula - a Language for Modular Multi-programming," Software - Practice and Experience, 7 April 1977.

[4]. Mitchell, J.G., Maybury, W., Sweet, R., "Mesa Language Manual," Technical Report CSL-79-3, Xerox Palo Alto Research Center, April 1979.

[5]. Wirth, N. "The Personal Computer Lilith," Institut fur Informatik, ETH Zurich, April 1981.

[6]. Brinch Hansen, P. "The Architecture of Concurrent Programs" Prentice-Hall, Englewood Cliffs, NJ, July 1977.

[7]. Redell, D.D., et. al., "Pilot: an Operating System for a Personal Computer," Communications of the ACM 23(2), February 1980.

[8] Stoneman, "Requirements for Ada Programming Support Environments," United States Department of Defense, February 1980.

THE NEED FOR PROCEDURES FOR KAPSE HOST OS UPGRADES

Herman Fischer and Herman Hess
(Litton Data Systems)

Summary

This paper addresses the least examined aspect of the Ada Environment, how to cope with the maintenance support of the Ada Programming Support Environment (APSE) Kernel (KAPSE) which interfaces the APSE tools to the underlying host operating system(OS). The paper proposes that KAPSES have an intimate relationship to each host environment OS; that host operating systems are upgraded often to fix bugs and improve performance; that a set of procedures be established to deal with the host OS upgrades; and that these procedures have legal and funding implications. The key points addressed in the following sections are:

The intimate relationship of KAPSEs to host OS's,
KAPSE characteristics,
Host OS characteristics,
Any host OS change requires KAPSE action,
Procedures to handle host OS upgrades,
Funding implications, and
legal implications.

The Intimate Relationship of KAPSEs to Host Operating Systems

APSE's, as derived from the DoD 'Stoneman', are usually depicted as in figure 1, as a set of tools supported by a Kernel which is in turn supported by a host operating system. Unlike the usual interactive user situation, where the host operating system provides the user interface to tools (e.g., from the inside of the circle outwards), in the ideal Ada environment, the APSE tools isolate the user from the underlying kernel and machine characteristics. As such, all of the operating system features which might be used by the Ada environment are handled by the middleman, the KAPSE. Since the KAPSE interfaces between APSE tools needing support services and the host operating systems providing these services, the KAPSE and the host operating systems enjoy a very close relationship.

The purpose of the KAPSE is to provide those support services which are normally provided by a host operating system, for the APSE programs. The KAPSE acts in the same manner that a traditional operating system would act, to the APSE programs, and thereby centralizes the host operating system dependencies. By centralizing these dependencies, a given APSE can be transported from one host computing system to another with changes localized to the KAPSE. This is the manner in which host-independent programs can be created for the DoD Ada environment. In addition, since the KAPSE provides its own user terminal support, the user view of the APSE is consistent between differing systems utilizing differing host operating systems.

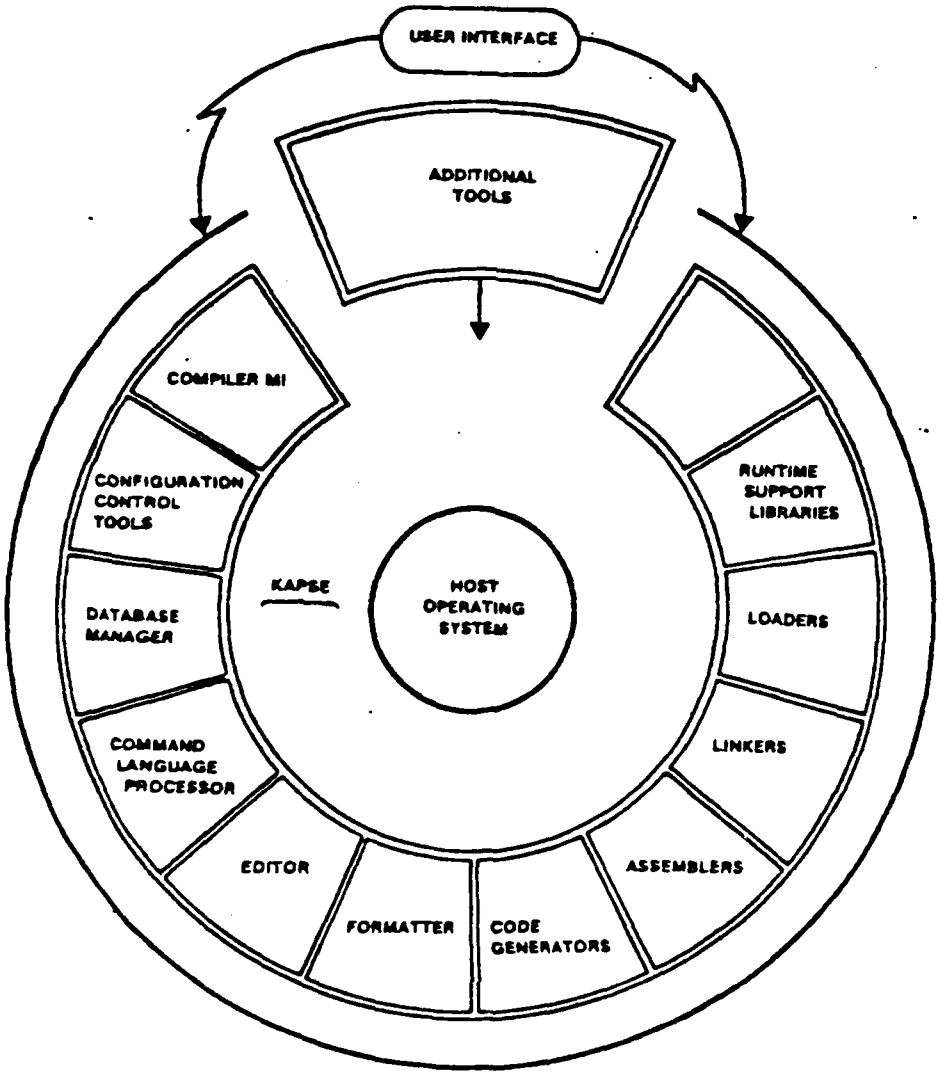


Figure 1. Typical depiction of APSE.

The KAPSE performs a number of sophisticated support functions which, in some cases, are implemented directly by calls to host operating system features, and in other cases are implemented within the framework of the KAPSE itself. The services which the KAPSE provides for APSE programs include:

lowest level of access to database,
Access to host file system,
Interprogram communication,
Program invocation and parameter passing,
Asynchronous control of program execution,
Support for host symbolic debugger and evaluation tools,
Host resource management,
Host system integrity features support, and
Host exception communications.

These services will be briefly addressed to provide a generic definition.

Each APSE must have a database of Ada compilation units and source code segments. These are to support independent compilation and 'WITH' clauses. The database must support storage of source code for multiple online users and for multiple developers of an integrated system of modules. This requires the existence of variants of source code, derivations, and associated configuration management data. Furthermore, as modules are compiled, execution objects are most likely stored in intermediate or executable forms along with sufficient data for linkage editing and debugging. The data may also include storage of data files to be used as test cases in exercising programs as well as storage of scenarios in the form of parameters to be passed to routines for bottom-up testing or to be returned by stubs in top-down testing.

The KAPSE must provide a limited and general form of access to the host operating system's file system. It must allow APSE programs and tools to manipulate sequential storage elements as well as to either provide within the KAPSE a directory or to utilize the directory services of the host operating system.

The KAPSE must provide a means of interprogram communication for the passing of data records between separate programs initiated by a single user as well as between independent programs initiated for independent on-line users.

The KAPSE must provide program invocation and parameter passing, such as for the initiation of APSE tools and for the initiation of Ada programs within the APSE environment. It may be desirable for the KAPSE to be able to initiate and pass parameters between Ada environment programs and standard utility programs of the host operating system.

The KAPSE must provide asynchronous control of program execution so that users may activate or discontinue the execution of tools, spooling procedures, offline printing, and executing job steps. Further more, the KAPSE must provide for support of program execution caused by the internal tasking structure of a given Ada environment program.

The KAPSE must provide support for a host symbolic debugger and evaluation tools. This facility allows Ada programs to be debugged and evaluated in the KAPSE and host environment independent of memory storage locations. (This is required because of unpredictability of execution location.)

The KAPSE must provide host resource management for those resources which are allocated in the host environment. These types of resources would include virtual storage, real memory storage, processing time, and management of those peripherals which might be dedicated to a program.

The KAPSE must provide support for integrity monitoring functions which are provided by the host operting system (and processor hardware). These are functions which assure the continued operation of the system hosting the APSE, including such activities as authorization and protection, security and classified data support, degraded operations detection (particularly where integrity may be compromized by hardware faiure), monitoring of performance data (runaway program loops), and monitoring of occurences of integrity-related exceptions (i.e., access violations, illegal operations, etc.).

The KAPSE must provide communication of host exceptions to the APSE. The host operating system, in notifying the KAPSE of exceptions, will indirectly notify the Ada programs of the Ada-generic equivalent of these exceptions. Typical exceptions would include arithmetic errors, invalid instruction operations, and constraint type checks, where implemented by the underlying host. Other types of exceptions which might be implemented include those which use virtual storage traps to provide segment loading and paging within the KAPSE structure.

Two examples of the relationship between a KAPSE and a host operating system are found in SofTech's planned Ada Language System(ALS) and Intermetrics' planned Ada Integrated Environment (AIE). Figures 2a and 2b show, for the ALS and AIE respectively, the planned interfaces between the host systems, KAPSES, users, and APSE tools and programs.

In the SofTech example, the host operating system is the Digital Equipment Corporation (DEC) VAX/VMS operating system. The KAPSE utilizes the VMS facilities for process initiation and termination. The KAPSE furthermore utilizes the host operating system for user command interpretation, memory management, and standard Input/Output (I/O). The standard VMS interprocess communications and control are utilized.

In the Intermetrics example, the host operating systems are the Perkin-Elmer OS/32 and IBM CP/370 VM. Although these two differ significantly in terms of resource management and program support services, the KAPSE is shown as primarily providing database operations, host and terminal I/O, and import/export for non-APSE systems and data. Those other forms of host OS support for the KAPSE, as found in the SofTech ALS, are not discussed in the AIE specifications.

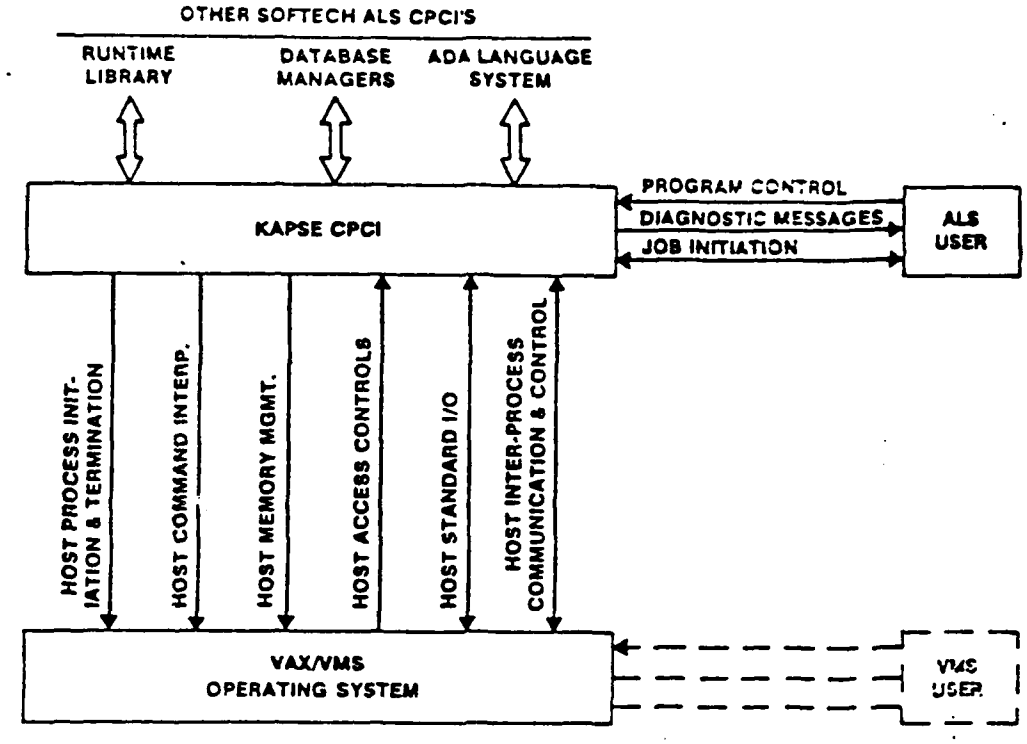


Figure 2a. KAPSE/Host OS Interface in SofTech Ada Language System.

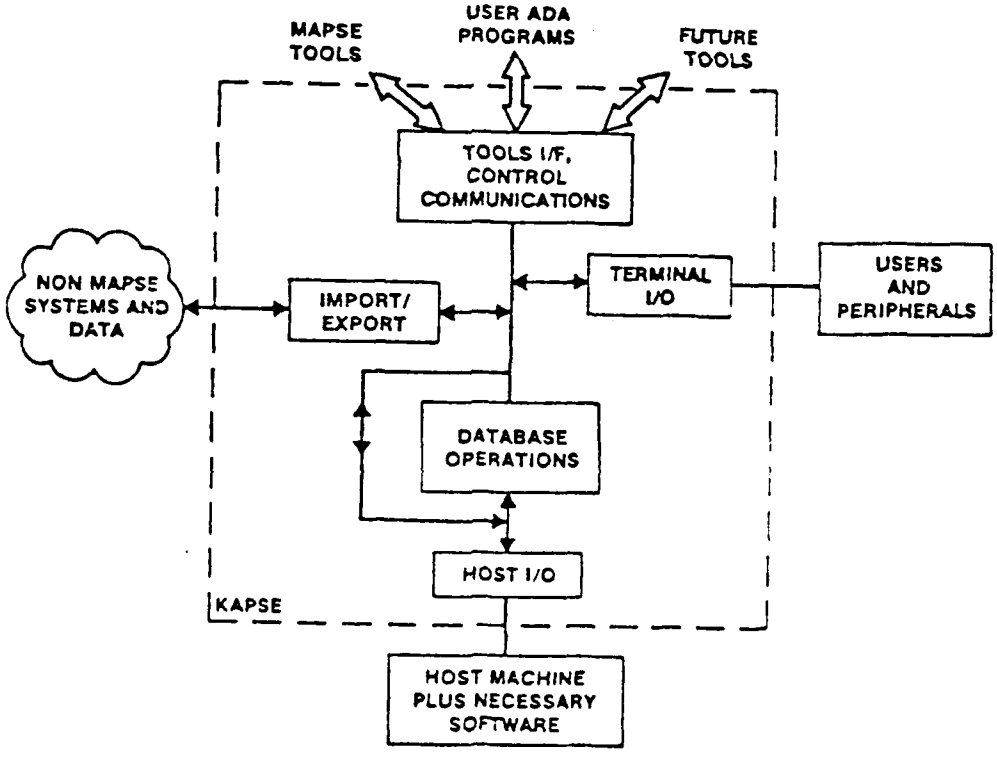


Figure 2b. KAPSE/Host OS Interface in Intermetrics Ada Integrated Environment.

In the SofTech ALS the user communicates with the KAPSE rather than with VMS as with non-Ada programs. By communicating with the KAPSE the ALS user can communicate in an Ada-unique command language which is then translated for the host's command interpreter, or directly acted upon by the KAPSE, as appropriate. Diagnostic messages from the host operating system are passed to the user by the KAPSE. The primary functions of the SofTech KAPSE are to implement the ALS database and a standard Ada command language for the ALS. The secondary purpose is to centralize all host operating system dependencies so by future substitution of KAPSEs, one may provide rehostability of the ALS.

Two Incompatible KAPSE Worlds

Referring again to the standard APSE diagram in figure 1, the domain of the KAPSE includes all of the services of the host operating system which may be needed by the APSE tools, as well as those additional features provided by the KAPSE itself. In examining the plans for Ada compilers in DoD and the Industry, one finds a wide spectrum of KAPSE approaches. These seem to cluster around what the authors will call the 'Ada View' and the 'Host OS View'. In the Ada view, the services which the KAPSE provides to the APSE tools are Ada-oriented. In the Host OS view, the services which the KAPSE presents to the APSE tools are host-oriented, in that host OS services are presented primarily intact to the APSE tools. Tools implemented for Ada-view KAPSEs are theoretically rehostable and transportable. Tools implemented for those KAPSEs which are primarily host OS views are very likely to require modifications when they are transported, due to a large dependency on host OS features. Let us examine the impact of this difference on file-type storage and on user command features.

In the Ada view of file storage, files are stored in what would be properly termed a 'database' rather than in file formats. The database directories for the access of storage would include Ada-related data such as derivation history, variants, and configuration management items. In the Ada view, the KAPSE would handle directories which are substantially different from those found in commercial host operating systems. The KAPSE would map these directories, and file storage and retrieval requests into low level input/output. The KAPSE would thus support access to files which the host's own programs would not be able to read. For this purpose additional APSE tools are required to implement utilities to map to and from host file formats. With the Ada-view of files implemented in a KAPSE, the file services, file formats, and potentially even the data storage representation files would be identical on any host system.

With the Host-OS view, files are stored in host OS formats. In this environment, a typical KAPSE would pass high level I/O requests from the APSE tools directly to the host OS itself. The host OS would then provide implementation of directory support and generate the low level I/O requests. This would allow the APSE

tools to use compatible directories and file formats with other host OS products. Utilities to provide file conversion would not be required. The KAPSE would provide some form of file renaming to allow for file generations, derivations, and variants. However, the tools of the APSE would not be freely transportable between different hosts because the services provided on each host operating system tend to be, in some manner, different from the high level file I/O provided on other operating systems.

In the Ada-view of user commmand languages, an interactive APSE user might sign on to his host operating system, which then would sign the user onto the APSE. Thenceforth the APSE would interpret all user commands and provide all responses to the user directly without host OS translation. This allows the KAPSE to map all the host OS commands used by the user into a standard Ada format, which is then independent of the host system. This also allows host exceptions to be mapped via the APSE into standard diagnostics and runtime messages for the Ada user. In the Ada-view of the command interface the use of the host operating system is restricted to those features supported by the APSE interface. For use of non-APSE host features, the user must log off the APSE and must then be knowledgable of the host's native command structure for native tools. The advantage of the Ada-view of command interfaces is that it is consistent between all hosts, provides the same terminal operation on all hosts, and that the user does not need retraining to use a rehosted APSE.

In the Host-OS view, the KAPSE is considerably simplified by not having a user command interface processor. With this view, the user interfaces with the APSE via the host operating system's command interpreter features. The host operating system's diagnostics and exceptions are preserved and used as with other host tools. The Ada language processor and APSE tools basically appear as additional products under the host operating system. The disadvantage of this system is that it requires the retraining of the users for the command structures, diagnostics, and exceptions handling for each host system. Furthermore, the operation of terminals, and screen controls are different with each system in this environment.

Figure 3 shows the three concentric circles of the APSE chart as represented in figure 1, but provides the additional dimension of depth for the wedges and rings in the circle. Three classes of KAPSEs are shown in figure 3. The first set shows an Ada-view APSE which uses differing KAPSEs to map a single set of APSE tools to multiple host operating systems. As can be seen, the KAPSE is significantly different for each host operating system. In the first example of the figure, the host operating system with fewer features requires a physically larger KAPSE to implement those features which are not found in the host system for the APSE tools. However the host operating system with significantly more features only requires translation of interface formats for the KAPSE to provide the remapped host functions to the APSE tools.

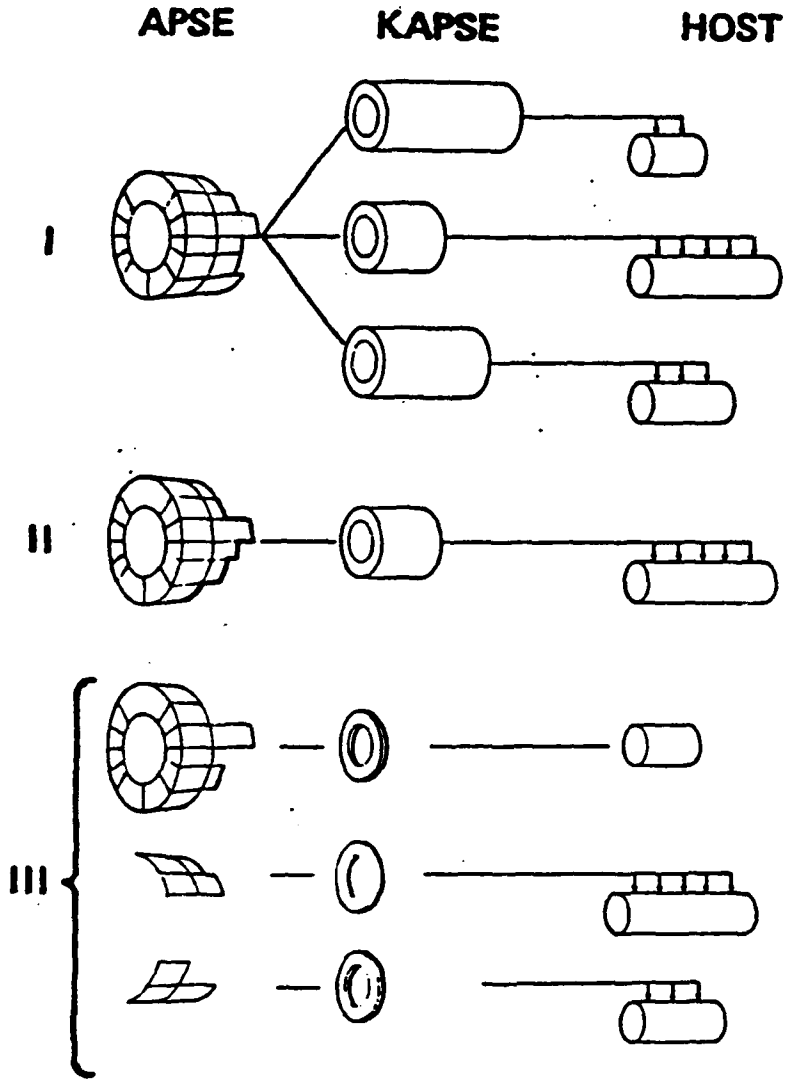


Figure 3. Three Cases of KAPSEs.

The second example of figure 3 shows the implementation of an APSE with a minimal KAPSE for a fairly sophisticated operating system that was not adequately defined for rehosting. In this case, the KAPSE and APSE closely relate to specific host operating system features. The KAPSE in such a case may provide implementation of a few additional features not natively provided by the host operating system, but does not provide translation from host operating system command format, file formats, and resource control to a generic interface.

The third example shows the case of a nominal KAPSE which basically presents host operating system services in their host view, directly to APSE tools. In the third case, the example shows that some of the tools of the APSE require substitution with different tools to correspond to the different features of an alternate set of KAPSE and host operating systems.

Host Operating Systems

This section will discuss characteristics of host operating systems that impact the maintenance requirements of APSE KAPSEs. The section will show that host operating systems are complex, that they are constantly evolving, and that they are never fully debugged. As a result, host operating system features and modules are subject to constant upgrade and to some degree, proliferation. The KAPSEs which interface these constantly changing host operating systems to a potentially stable set of APSE tools will need to be modified and upgraded as the host operating systems themselves are modified and upgraded. This will generate the requirements for KAPSE upgrade procedures which are discussed in a subsequent section.

Host operating systems are today very complex. The size of host operating systems encountered by the authors has varied from 1/2 to ten million lines of source code in assembly language, or more recently, in languages such as 'C'. Host operating systems include, as they are distributed and generated, many language compilers and utility programs. Disk storage control and database implementations include elaborate directory structures and comprehensive access methods. For multiuser computer systems, host operating systems provide sophisticated forms of resource control and often virtual management of processor resources.

Host operating systems appear, in the authors' opinion, to be constantly evolving. One of the prime reasons for this constant change is the addition of new features to each operating system so that it may stay competitive. Frequently, competitiveness is determined by the operating system's performance on a given set of hardware. Performance is often enhanced by new features, improved file access methods, new utility programs, and concepts of being friendly to users. Evolution in operating systems has a secondary reason, namely, the support of a constant and consistent evolution in computing hardware. Generally new CPU and associated memory hardware appears every five years of less, for most manufacturers

of host operating systems. New communications for support of local and remote users appears even more often than that. The support of new hardware generally causes the release of new operating system versions and often the addition of new operating system features to take advantage of improved hardware technology.

The final reason for the evolution of operating systems is the response of the host system vendor and independent operating system vendor to system problems regardless of the actual source of the problem. Owing to the difficulties of using sophisticated operating systems and to the proliferations of terminal facilities and mini-computers, has been the increase in quantity of casual users with insufficient information and knowledge of system requirements for satisfactory use. There exists a demand that systems become more friendly; it is now unacceptable for a system to shut down because an error has occured regardless of the source. Hence, operating systems like UNIX (tm) have required numerous changes to increase reliability of operation. Well-established operating system products like the IBM OS/370 (with its numerous variants) have required the introduction of products to enable users to find a "friendly" environment (such as SPF). The impact of the trend to make operating systems friendly and usable has been that revisions and fixes of existing operating systems have cascaded into systems which have never been fully debugged and which, with available resources, are not likely to be ever error free.

The authors believe that host operating systems are never fully debugged because of three reasons: (1) they are large and complex, (2) they are subject to constant evolution, and (3) Murphy's law.

The significant size of every multiuser sophisticated operating system prevents its total comprehension by any single programmer or by any single set of gurus at a vendor's facility. Thus an operating system is normally a team effort even in the support phases. As members of the team change, as people move to new positions, and as internal political pressures change, inconsistencies in implementation and oversights occur. This is a natural cause of bugs.

As discussed earlier in the section, operating systems are subject to constant evolution driven by hardware technology enhancements and driven by the addition of new features to enhance performance and competitiveness. The constant evolution causes constant changes and whenever as large a collection of modules as an operating system is changing, new bugs are introduced to the system.

The final reason why the authors allege that operating systems are never fully debugged is because of what is commonly known as 'Murphy's Law'. The types of bugs and defects extant in an operating system occur in areas where they are least expected. These areas are least well tested in the process of releasing new versions of the operating systems.

Table I is provided to show the large category of features in an operating system which are potentially used by a KAPSE. In Table I, the features enumerated vary greatly in the mean time between major updates of these features. For example, terminal communications features for local networks are presently being frequently updated due to newly emerging technology. On the other hand, local direct connection of terminals is in general a stable feature. Mass storage support and user interface support include features which are hardware technology driven and performance competiveness driven to cause frequent major upgrades in these areas. The system control services have become an expanding area which, for military applications and embedded computers, have become a requirement.

Table II is provided to show the proliferation of variants in host operating systems. For example, in both the IBM and DEC worlds, a significant number of operating system variants exist for the same processors. While several operating systems in the table have similar interprocess communications and resource control features (such as the UNIX systems), others (even on the same hardware) are significantly different (such as IBM OS/VS from IBM VM, or VAX VMS from VAX UNIX). APSE tools and programs are likely to be hosted on any number of combinations of these operating systems and thus the problem of KAPSE updates is further aggravated by the large number of KAPSE variants required. For example, on the DEC VAX computer, one may well be expected to support one of the listed three operating system combinations, depending on a user's actual site configuration. It is likely that a user with one of the operating systems already installed will be unable to switch to another combination without significantly impacting his other programs and procedures.

Procedures for Host Operating System Upgrades

This section develops a procedure for the handling of KAPSE upgrades in response to upgrades in underlying host operating systems. The simplifying assumption is temporarily made that Ada changes and problems will be controlled and incorporation into this procedure will not be a problem.

A cursorary examination is first made of some of the salient characteristics of the vendor process of upgrading the host operating systems:

(1) Operating systems(OS) are developed initially by independent parties. The developer's procedures which result in releases of OS upgrades are be characterized by the following observations:

a) The scheduled release date for usage of an upgrade may be delayed. The developer assumes no liability for scheduled delivery failures.
b) The documentation and installation instructions are likely to be incomplete and incorrect during early development.

TABLE I.

HOST-OS FEATURES SUBJECT TO UPGRADE

FEATURE POTENTIALLY USED BY KAPSE

TERMINAL COMMUNICATIONS
- LOCAL CONNECTION — DIRECT
- LOCAL NETWORKS
- REMOTE CONNECTIONS — DIRECT
- REMOTE NETWORKS

MASS STORAGE
- MEDIA AND DEVICE HARDWARE
- DEVICE I/O HANDLERS
- FILE DIRECTORIES AND ACCESS METHODS
- UTILITIES

USER INTERFACE
- COMMAND INTERPRETATION
- SOURCE EDITORS
- FILE MANIPULATION UTILITIES
- PROGRAM INITIATION

RESOURCE CONTROL
- MEMORY ALLOCATION
- PROCESS/TASK CONTROL
- INTER-PROCESS COMMUNICATION

SYSTEM PROCESSORS
- LANGUAGE PROCESSORS
- LINKERS
- DEBUGGERS
- RUN-TIME LIBRARIES

SYSTEM CONTROL
- PERFORMANCE MONITORING
- ACCOUNTING & EVALUATION
- DIAGNOSTIC SERVICES
- SECURITY OF OPERATIONS
- INTEGRITY FEATURES

TABLE II.

PROLIFERATIONS OF HOST OPERATING SYSTEMS

IBM WORLD	DEC WORLD	NON-DEC NON-IBM WORLD
MVT	PDP-11 IAS	UNIX VS. 7
MVS	PDP-11 DOS	UNIX SYSTEM III
VM	PDP-11 UNIX	CP/M
AMDAHL	VAX-11 VMS	1100/OS
NSS	VAX-11 VMS/UNIX	6600/7600
	VAX-11 UNIX	SHARE/7
	DEC 10 TOPS-20	MCF (?)

c) The announced release may not be available for independent evaluations and review before release date.
d) When an announced release has been permitted to be reviewed or evaluated the using community's recommendations for revision or correction is still only a recommendation.
e) The life period of the release is subject to termination of developer support, which may be announced without user consideration.
f) The interface and performance characteristics of separate and succeeding OS releases are subject to changes which may not be readily apparent.
g) The OS release may not have a specification standard which is under configuration control nor need it necessarily abide by published standards which may exist.
h) The OS developer is a disinterested party with respect to Ada support. His primary revenues come from large volume machine sales/installations, generally to Cobol, Basic, or 'C' business type users.

(2) The interface standards for the APSE tools (e.g., Ada Translator) to KAPSE facilities can be defined, controlled, and can be scheduled subject to changes in Ada specifications and requirements.

(3) An important task will therefore be a requirement to specify a procedure to control and verify requirements and candidate implementations of the KAPSE-to-OS interface.

Based upon the characteristics stated above, which imply that OS maintenance from the Ada viewpoint is essentially uncontrolled, a requirement exists to create a procedure for control that is both responsive and flexible. The assumptions are essentially unchallangable. The latest example of significant magnitude is the recent history of UNIX development, e.g., each vendor provides his own version of UNIX despite the common source from Western Electric.

The question arises as to whether all the changes are necessary and could they be reduced if more careful implementation methods were employed. It should be clear that while the number of changes required because of coding and logical design errors could be reduced, the reason for changes comes from other causes. Namely, the need to accomodate new characteristics of the CPU and available peripherals which require inclusion to the OS for service, and the need to adjust to local (user) impact of user applications. Since OS's are very large and complex, it should be apparent that a control procedure has to recognize the need to modify and correct and will continue to be a maintenance item.

Briefly the following procedure can be employed as a baseline procedure subject to minor modifications for recognizing incremental and incomplete upgrades.

Using the figure given below the procedure can be outlined as follws:

Procedure Step	Operating System	KAPSE	Baseline
1.	Vx	K0	Initial starting point
2.	Vx.1	K0	Initial trial baseline
3.	Vx.n	K0.1	1st Certified Release
4.	Vx+1	K0.1	Modified OS has been released
5.	Vx+1	K1	Proposed new baseline

Procedure step 1: Select Operating System (OS) which is a candidate baseline for interfacing the KAPSE where Vx is the identification of the OS and K0 designates the KAPSE.

Procedure step 2:
a) Determine modifications and changes to OS Vx to interface with the KAPSE.
b) Perform a verification and maintenance procedure of the interfacing system until a predetermined level of performance has been attained. This version of the OS is identified as Vx.1 and is provisionally released for selected trial evaluations.

Procedure step 3:
a) Establish a Change Control Board (CCB) to determine the needs and priorities for proposed revisions and changes to the APSE. (This is a problem area because of conflicting requirements and resources among users as to the use of funds and apparent priorities.)
b) Based upon the results of procedure step 2b and procedure step 3a, determine which revisions and changes shall be implemented.
c) Implement and verify all approved changes that were scheduled for implementation.
d) Designate OS Vx.n and KAPSE K0.1 as a certified release where the subscript qualifiers .n and .1 are used as constants for identification purposes.

Procedure step 4:
a) A new OS version has been released where x+1 represents a successor OS which the releasing agency announces as a significant change.
b) Evaluate the effects of OS Vx+1 on KAPSE K0.1 to determine the need for revisions.

Procedure step 5:
a) Designate KAPSE K1 as the candidate for interfacing to OS Vx+1.
b) Submit proposed changes to the CCB for processing. (This is also a problem area in which a policy of maintenance must be established. It not only includes the evaluation of needs but also requires obtaining and allocating resources.)
c) Return to procedure step 2b and procedure 3 to continue maintenance.

Funding Implications

The implications of the above procedure on funding required to support DoD and private-industry APSE KAPSEs is especially severe. One is faced not just with a single operating system on a given popular host, but with a multiplicity of systems. Then, in addition, there are a number of popular host systems, with each DoD service supporting several.

Today, there are only two DoD-sponsored APSE efforts, the SofTech ALS hosted only on the VAX under VMS, and (reportedly) the Intermetrics AIE hosted on the IBM 370 only under VM and on the Perkin-Elmer under OS/32. It is inconcievable that the ALS won't appear also on VAX UNIX, just as one would expect the AIE to appear on the IBM 370 also under VS. DoD funded maintenance efforts for these systems will have to include, practically in perpetuity, CCB's for both efforts with significant funding to track all of the differing KAPSE versions and retest/recertify each one as it's corresponding host OS is upgraded.

Furthermore, there are a number of additional APSE and Ada efforts. While not sponsored by the DoD directly, a number of these efforts are "sanctioned" for one reason or another (such as the ones which generate microcomputer-chip object code). Pressures of scheduling, time, money, and politics, will cause these other Ada efforts to be utilized in weapons and defense-type systems. In order to accept one of these "other" Ada efforts, a project manager should consider the impact on the maintenance costs of his system of having to sustain applications code and possibly to be forced into the KAPSE maintenance business. Possibly, when a non-sponsored Ada effort becomes important to the interests of the national defense, DoD should have a procedure for contributing to the maintenance of these Ada efforts from common funding. (There is no clear-cut solution offered for this problem, only the advice that it must be acknowledged.)

Legal Implications

A close relationship between the KAPSE CCB's and maintenance teams, and those parties which supply and maintain the host operating systems, is necessary for many reasons. Primarily, a close relationship minimizes the trauma of time gaps between new upgrades of a host operating system and the corresponding release of a recertified matching KAPSE. However, such a close relationship requires the host vendor to disclose all of his plans to the APSE CCB and maintenance community. In the commercial world, secrecy has been associated with new product developments to maintain competitive edges. The authors do not know how an APSE CCB could enter to nondisclosure agreements, in good faith, with competing vendors of host equipment, and be able to control the inadvertant leaking of industrial secrets. Just the normal migration of personnel in the community, such as from the CCB team to one of the competing vendors, could be a problem.

A FORMAL APPROACH TO APSE PORTABILITY

R. S. Freedman
(Hazeltine Corporation)

Introduction

We are interested in describing the portability of host independent Ada programs that call a KAPSE. These programs utilize KAPSE services (such as the CALL mechanism or the database manipulation facilities) where recompilation is insufficient to insure portability. It is therefore important to understand how a KAPSE interfaces with its "users" and how a KAPSE organizes its database. We examine the concepts of "portability," "APSE equivalence," and "minimum toolset," in a context of formal syntax and semantics. These issues are discussed informally in (6).

Syntactic Issues

In this section, a KAPSE is modeled as a language (a set of lexical elements) generated from a grammar. This grammar consists of the grammars of Ada and the command language (which also contains the grammar of the database). If two KAPSE's have the same grammar, then they have the same syntactic (phrase) structure in their command languages and databases (the Ada syntax is already standardized). In this case, the problem of tool portability is reduced to standardizing KAPSE lexical units (names in the Ada source code, command language, database, and user interfaces).

If two KAPSE's have different grammars, then we do not know if the languages generated by these two grammars (the corresponding KAPSE's) are equivalent in phrase structure. This problem is unsolvable if the different grammars are context sensitive. In other words, if the syntax of two different command languages or database schemas (2) are specified as context sensitive grammars, no algorithm exists that can determine whether the resultant languages (legal sequences of lexical units or legal database instances) are equivalent.

If the two KAPSE's are generated from different grammars that are regular, then the problem of determining KAPSE equivalence is solvable. Unfortunately, another unsolvable difficulty exists if we do not standardize on one syntax defined translation from KAPSE to KAPSE. No algorithm exists that determines whether two syntax defined translations (from a regular grammar to a regular grammar) have equivalent "dictionaries" (1).

Semantic Issues

Concepts from formal semantics (4, 5) are used in this section to model the tool portability problem. We model a KAPSE database as an abstract semantic domain. This semantic domain D contains the effects of calls to the KAPSE (as far as storage allocation and updating is concerned). Calls to the KAPSE will occur in some syntactic domain K. The effect of a KAPSE call is to create an automorphism on the semantic domain. Consequently, a KAPSE call C is a semantic function from K to the set of all automorphisms from D to D:

C: K $\longrightarrow$ Aut (D,D) or in the notations of (4),

C: K $\longrightarrow$ D $\longrightarrow$ D

where

K is the syntactic domain of KAPSE calls (following the syntax of Ada), $k \in K$ is a syntactic element (a particular call to the KAPSE),

D is the semantic domain of the KAPSE that contains all "meaningful" information relative to the effects of a KAPSE call, and

$d \in D$ is a semantic element (a particular state of the KAPSE domain) sometimes called a store.

An automorphism from D to D is an invertible transformation (isomorphism) from D to D (what can be done can be undone).

An automorphism T on the semantic domain is denoted by

$$T = C[\![k]\!]$$

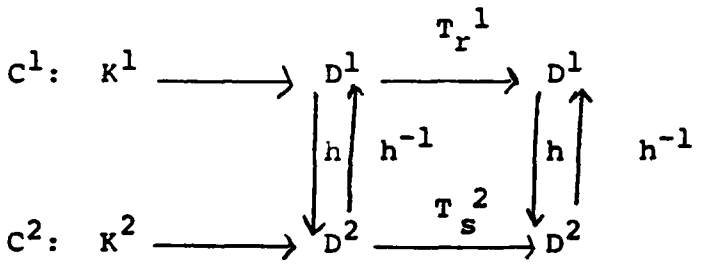
where we (traditionally) use the square brackets to denote syntactic elements. T and k both can be thought of as (KAPSE)"tools;" T conveys semantic information and k conveys syntactic information.

The "result" of executing call k relative to a particular store d is therefore

$$T(d) = C[\![k]\!]d$$

We model a KAPSE as the triple (C,K,D).

We now assume that we have two different KAPSE's (C^1, K^1, D^1) and (C^2, K^2, D^2). We say that these two KAPSE's are equivalent if there exists an isomorphism h from D^1 to D^2 such that the following diagram commutes:



In other words, two KAPSE's are equivalent if for any two calls k_r^1 and k_s^2

$$C^1 [\![k_r^2]\!] h = T_r^1 h = h T_s^2 = h C^2 [\![k_s^2]\!]$$

This isomorphism h is an isomorphism between "meanings" or semantic domains. Essentially it allows any tool T_r^1 in (C^1, K^1, D^1) to be written in terms of its "corresponding" tool T_s^2 in (C^2, K^2, D^2).

In other words, if KAPSE command k_r^1 is utilized but KAPSE (C^1, K^1, D^1) is not available, k_r^1 can be utilized on KAPSE (C^2, K^2, D^2) if h exists. In this case, k_r^1 is replaced by a "virtual KAPSE command," or semantically

$$T_r^1 = h T_s^2 h^{-1}$$

The isomorphism h is similar to a "gateway protocol" between different communications networks. In our case, h is a protocol between KAPSE functions and databases.

We can use some syntactic information in K to help assess the semantic "differences" in KAPSE structures. This information can provide a metric for statistical inference. One method described in (2) develops a certain function related to the sequence and use

of the syntactic elements; this function is related to the information theoretic "entropy" and can be considered to be (statistically) invariant from KAPSE to KAPSE.

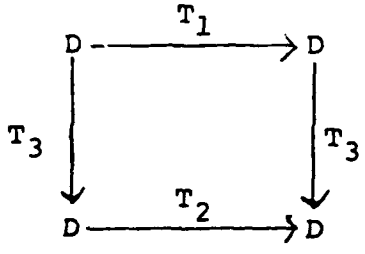
"Minimal Toolsets"

The above semantic model can be extended to APSE tools. An APSE call S is a semantic function

$$S: A \rightarrow D \rightarrow D$$

where A is the syntactic domain of APSE (in Ada syntax), and D is the semantic domain of the APSE. A (semantic) tool T is denoted by a function $S[\![a]\!]$.

Two APSE tools T_1, T_2 are equivalent if there exists an (automorphism) tool T_3 such that the following diagram commutes:



Essentially two tools are equivalent if one can be constructed in terms of the other.

A generic tooset is a set of tools such that any two tools are equivalent. A generic toolset is an equivalence class of tools. An APSE with no generic toolsets is thus a "minimal" toolset: no tool can be constructed out of any other.

Conclusion

Aspects of formal syntax and semantics have been applied to the problem of tool portability. Formal syntax considerations show that no algorithm exists that can determine whether two KAPSE's (specified by two different context-free grammars) are equivalent. Formal semantics was used to model calls to a KAPSE as a function that maps a syntactic structure to a domain of meanings (a semantic domain). The problem of semantic equivalence of two KAPSE's is reduced to the problem of constructing a virtual KAPSE command. The virtual command utilizes a special tool that acts as a gateway between KAPSE semantic domains. A (machine-independent) APSE tool is portable if its calls to its KAPSE can be replaced by virtual calls to another KAPSE.

References

(1) Aho, A., Ullman, J., The Theory of Parsing, Translation, and Compiling, Volume I, Prentice-Hall, Englewood Cliffs, 1972, Chapter 3.

(2) Fleck, A. C., "Towards a Theory of Data Structures," J. Computer and System Sciences, 14, October 1971, pp. 617-627.

(3) Freedman, R., "A New Approach to Software Science" (to appear).

(4) Scott, D., Strachey, C., "Toward a Mathematical Semantics for Computer Languages," Proceedings of the Symposium on Computers and Automata, Polytechnic Institute of Brooklyn Press, New York, 1971, pp. 19-46.

(5) Tennent, R., Principles of Programming Languages, Prentice Hall International, Englewood Cliffs, 1981.

(6) United Kingdom Ada Study Final Technical Report, Department of Industry, London, 1981, Volumes II, VII.

AJPO KIT POSITION PAPER

Anthony Gargaro
(Computer Science Corporation)

A significant challenge to be overcome in achieving the economic and efficient transportation of tools and data bases among the APSEs is the development of KAPSE interfaces that provide Capacity Transparency. Capacity Transparency implies that the KAPSE abstracts host system capabilities into its interface semantics, where appropriate and feasible, without compromising the transportability of the interfaces. The semantics of interfaces developed in this manner allow exploitation of host capabilities that are not always guaranteed on every system.

This refines the derivation of interoperability and transportability properties, and motivates a thorough understanding of both tool and data base requirements, in conjunction with an appreciation of the candidate host system capabilities that interact with these properties. An APSE-KAPSE interface may then be conceived that is semantically neither permissive nor overly restrictive. Encapsulation of these properties into the KAPSE through interface templates that are available for a capability taxonomy of potential host systems provides a straightforward paradigm for a generic virtual operating system. For example, a processing resource taxonomy might include uniprocessors, multiprocessors, and distributed processors (Figure-1).

One interface where substantial difficulties are anticipated in deriving transportability properties is the interface that specifies code execution services. These difficulties arise in part from the inconsistencies and diverse functionality of interfaces in existing KAPSE designs. In addition, the interfaces are often conceived without recognizing the need to flavor the interface semantics with processing resources of the host system because they are founded on host facilities that multiplex concurrent code executions on a single processor.

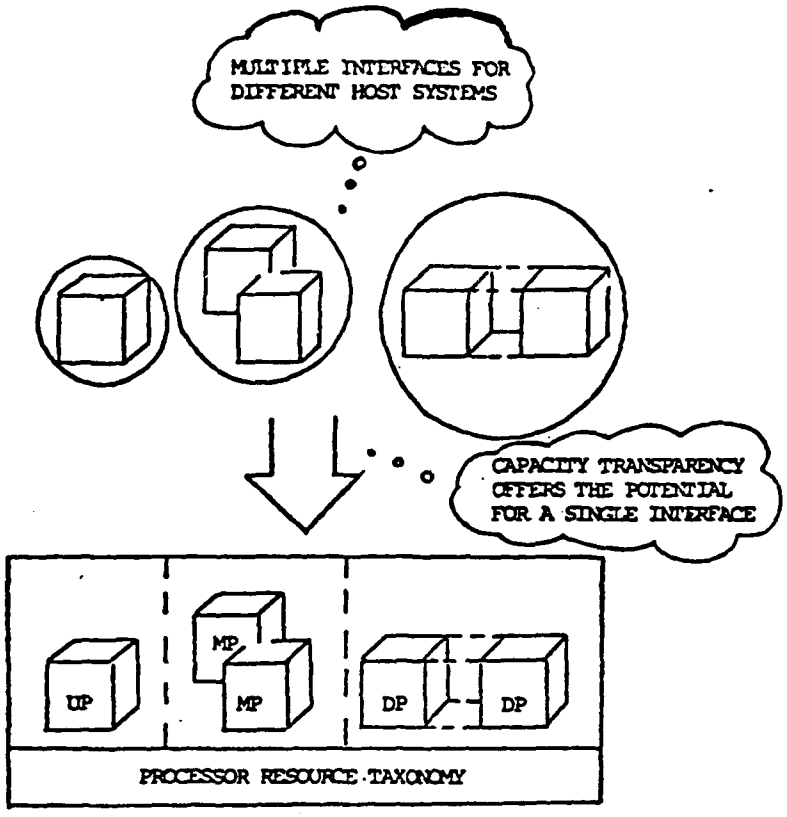


FIGURE-1 : CAPACITY TRANSPARENCY OF VARIOUS HOST ENVIRONMENTS MOTIVATES IMPROVED TOOL TRANSPORTABILITY.

A proposed area of study is the comprehensive formulation of mandatory tool requirements to be satisfied by the code execution services available through the APSE-KAPSE interface. Currently, the orientation for these services has resulted in their fragmentation into a number of different interfaces. This may be attributed to the placing of design emphasis on the functionality of the requesting tool, rather than on the functionality of the interface. Consequently, the interface reflects a semantic bias towards the tools, which are designed to be insulated by this interface from the actual processing resources of the host systems. Therefore, the interfaces tend to ignore Capacity Transparency, a feature that would increase transportability properties.

An important objective of the proposed study is to research a unified interface that encapsulates all facets conveyed by the notion of code execution. This notion should combine the desired user perceived operational characteristics of an APSE with those that may be acquired from the host system. An additional objective is to explore an improved degree of transportability for those components of the APSE that use code execution services; for example, the Ada runtime system.

The code execution services of the KAPSE incorporate the facilities that manage and maintain the execution of separate, independent and dependent, threads of control (code executions) within the APSE. The fundamental capability to support separate threads of control is required by the explicit and implicit demands of the Ada language[1,2] and the APSE[3,4]. In the Ada language, the task model and parameterized main programs provide both explicit and implicit requirements for separate threads of control, while in the APSE, separate threads of control are conducive, if not essential, to tool invocation and multiuser operation. At a minimum, this service interface must support the code execution requirements of the Command Language Processor and the Debugger (Figure-2).

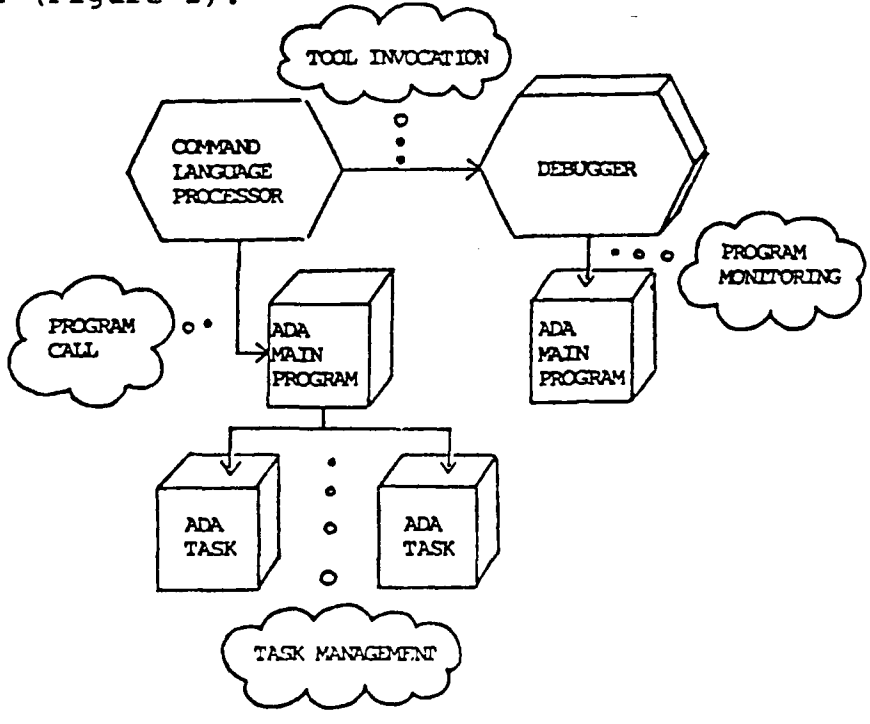


FIGURE-2: CODE EXECUTION SERVICES SUPPORT TOOL INVOCATION PROGRAM CALL AND MONITORING, AND TASK MANAGEMENT.

In order to specify a schema for these services, it is first necessary to develop a conceptual framework that represents a coherent synthesis of the appropriate semantic domains. One contributing factor to the absence of a currently acceptable framework is that the execution (and communication) semantics of an Ada main program are beyond the province of the language definition. Therefore, the semantics have been strongly influenced by the specific requirements of the tools committed to support the APSE, and by the initial host system facilities that have been available. Evidence of this approach is manifested in some of the early APSE designs that have been published for review. All the designs furnish similar facilities, however none seem to guarantee that the specified interface is suited to differing host systems. In addition, some of the designs have arbitrarily divorced support for the Ada task semantics from the program control services for the tools. This dichotomy would seem to be detrimental to the overall transportability of the APSE.

While specific recommendations are outside the scope of this position paper, some suggestions are pertinent to guide further research in developing a framework based upon the previous observations and the premise for Capacity Transparency. These include:

1. Defining the semantics for main program execution and communication
2. Specifying the requirements for synchronous and asynchronous program execution
3. Defining the relationship and dependencies that derive from tool invocation and program call
4. Investigating the integration for Ada task support in the code execution services of the APSE-KAPSE interface
5. Specifying the requirements for program monitoring
6. Investigating the optimization of various host system processing resources through Capacity Transparency.

The four government funded APSE design efforts have specified proposed interfaces between the APSE and KAPSE. The interfaces for the above code execution services, as developed for these designs, have

varying styles of functionality [6..9] (Figure-3). In order to establish a meaningful framework, the KAPSE for the Ada Language System (ALS) design is used as a base of reference.

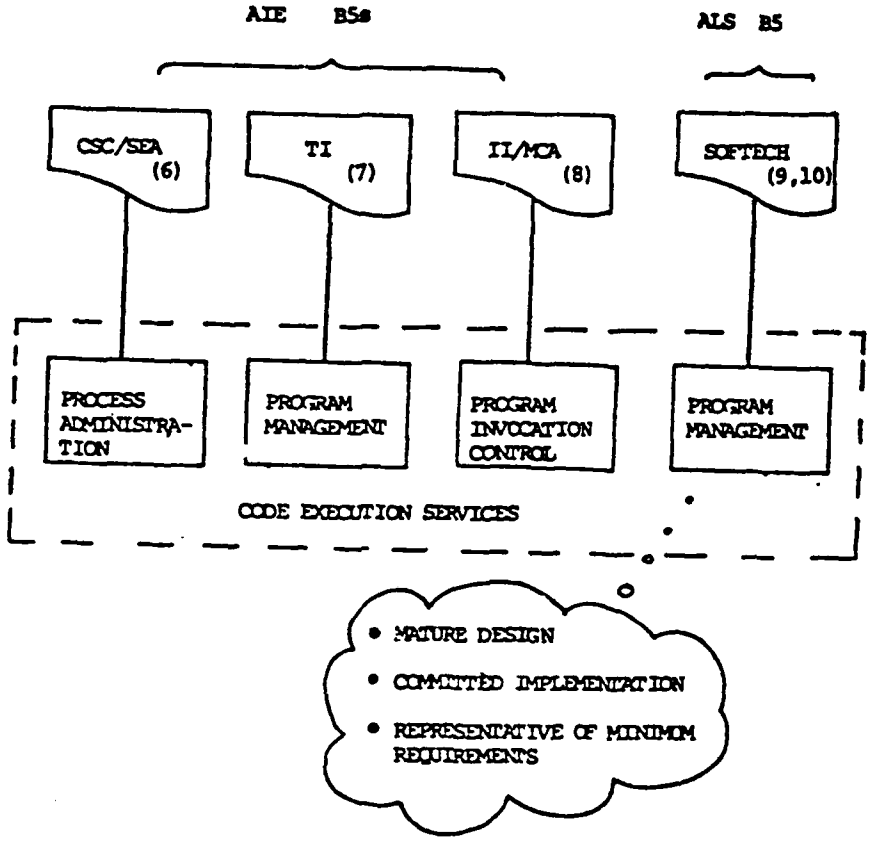


FIGURE-3 : REFERENCE TO A PROPOSED KAPSE INTERFACE PROVIDES INSIGHT INTO PROBLEM DOMAIN.

Although these designs have undergone extensive evaluation [5], the referenced design is the one that promises to achieve an implementation status in the near future. This choice does not necessarily indicate that its APSE-KAPSE interface is preferred to those of the other designs, but merely recognizes that because of implementation chronology, it may become a defacto standard.

To gain an appreciation of this design, an informal synopsis of the salient interfaces is presented. At the outset, it should be noted that tool transportability was not the primary requirement of this early design effort.

The principal, currently defined, code execution services are provided through the facilities of four functional domains that constitute the Program Management interface within the KAPSE (Figure-4). Only two of these domains encapsulate services that are available through the APSE-KAPSE interface. These are the Program Call and Program Control services. The semantics of these services permit an Ada main program to execute as an independent thread of control in the host environment. Multiple main programs may execute concurrently for a user in the context of an APSE job.

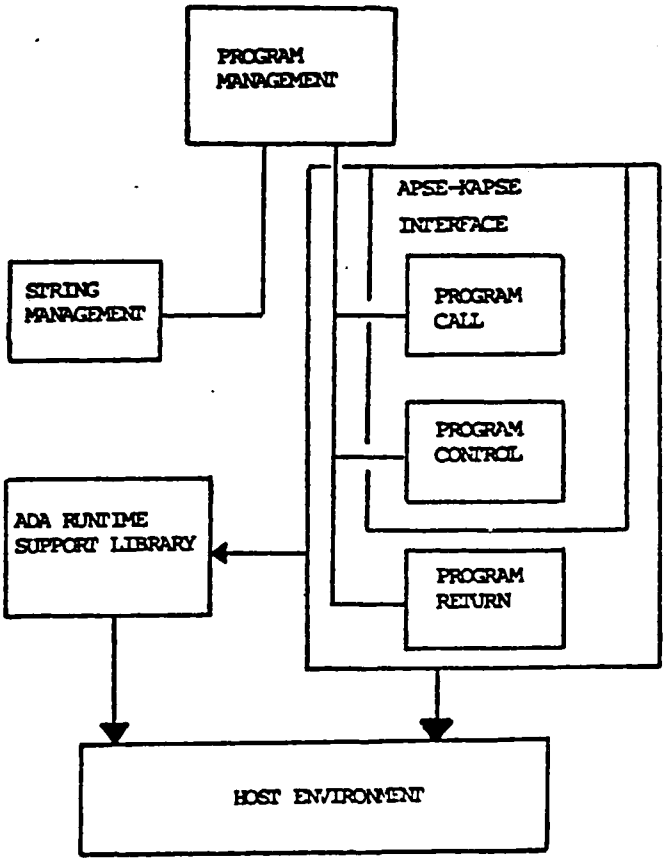


FIGURE-4 : PROGRAM MANAGEMENT ENCAPSULATES APSE-KAPSE INTERFACES FOR CODE EXECUTION SERVICES.

A CALL tree is used by the KAPSE to maintain the call relationships of programs within a job. The Program Call interface includes services to call (invoke) a program and await its completion, and to call a program and to continue execution. These services are provided to a user through the Command Language to request the execution of a tool (Figure-5). The semantics of each call allows programs to exchange information through formal parameters and predefined string variables.

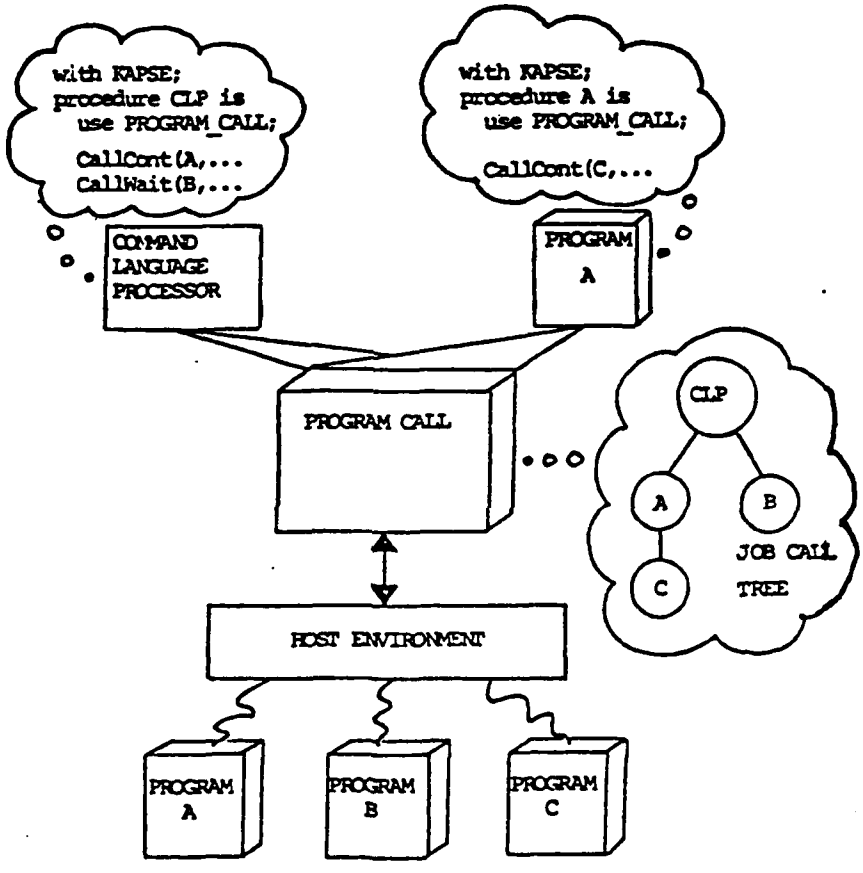


FIGURE-5 : PROGRAM CALL SERVICES ALLOW PROGRAMS TO EXECUTE AS INTERLEAVED, PARALLEL, OR DISTRIBUTED THREADS.

The Program Control interface includes services for accessing and updating program status information, requesting program abortion, and for requesting program resumption. These services are invoked by a user through an APSE tool. The semantics of each service request

are constrained using the CALL tree context (CALL node) of the calling program, or that associated with a program specified in the request. For example, a request to terminate a program is defined only for a program that has descendant programs in the CALL tree.

As a result, these combined services constitute a modest, reasonable, and transportable interface that can be implemented on a number of contemporary host systems. However, upon more detailed scrutiny of the interface specification and the attendant functional composition for its implementation, some undesirable trade-offs have been noted. These trade-offs preclude acceptance of this interface as an appropriate model for future development, if improved Capacity Transparency is to be pursued.

A simple model of code execution services can be derived by abstracting the semantics of the two interface specifications. Briefly, the KAPSE provides separate threads of control for Ada main programs.

Programs may call and communicate with other main programs, and as a result establish a CALL tree of program call nodes. Programs may perform limited control over the execution of descendant programs. Programs may execute serially interleaved on a single processor or in parallel on multiple processors. Execution on distributed processors is not excluded by the interface specifications, although program communication would appear to be more oriented to tightly-coupled (shared memory) systems. Reconciliation of the threads of control to the processing resources is the responsibility of the KAPSE.

One of the potential flaws of this APSE-KAPSE interface is that it does not encapsulate services for code execution below that of an Ada main program. Therefore, services for task execution have to be accomplished by some other functional domain. In the referenced model the Runtime Support Library (RSL) includes task execution facilities that comply with the Ada task model [10]. The tasking facilities are classified as Task Management and Task Dispatching. These facilities are permitted to directly request host system facilities, thereby avoiding the APSE-KAPSE interface (Figure-6). As a consequence, the transportability of the RSL is compromised.

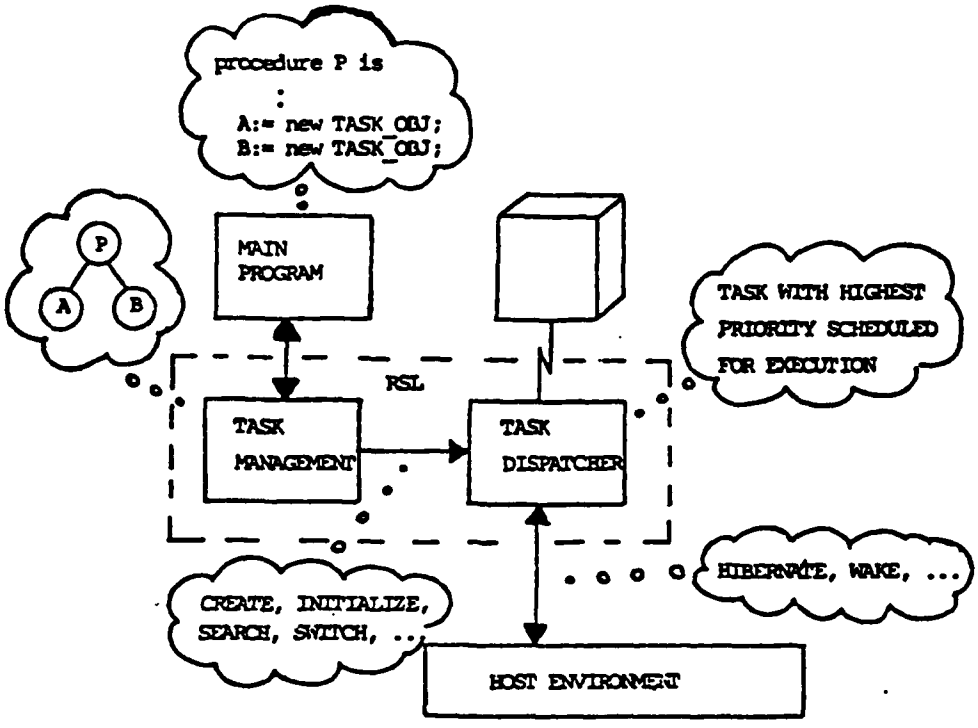


FIGURE-6 : ENCAPSULATION OF LOW LEVEL EXECUTION SERVICES IN THE RSL CONSTRAINS CAPACITY TRANSPARENCY.

Another deficiency occurs because the high level program execution interface of the APSE-KAPSE requires that the Task Dispatcher must execute tasks as serially interleaved code executions. This is contrary to the concept of Capacity Transparency, since on multi-processors such as the Intel iAPX 432 tasks should not be constrained from parallel execution. Ideally, the RSL should be made transportable by requesting all host facilities through the APSE-KAPSE interface. Task level execution services are then made an essential requirement of the interface, although the majority of task management is retained in the RSL.

From these observations, the existing code execution model would be revised to conform with the transportability rules that have been recommended. In this revision it is assumed that the currently defined interface semantics are adequate to support the Ada task model. The primary revision is a redistribution of functionality through the relocation of task dispatching from the RSL to the KAPSE. The interface

between the Task Management and Task Dispatcher now becomes a part of the APSE-KAPSE interface (Figure-7). This redistribution warrants careful review of the existing interface in order to promote a Capacity Transparent interface, and to accommodate the additional complexity of having task dispatching execute within the KAPSE. For example, CURRENT_TASK is now required to enable multiple task identifications, and the task suspension interface between the KAPSE and RSL for program invocation becomes obsolete. In fact the semantics of call program and wait would have to be modified to indicate the nature of the wait.

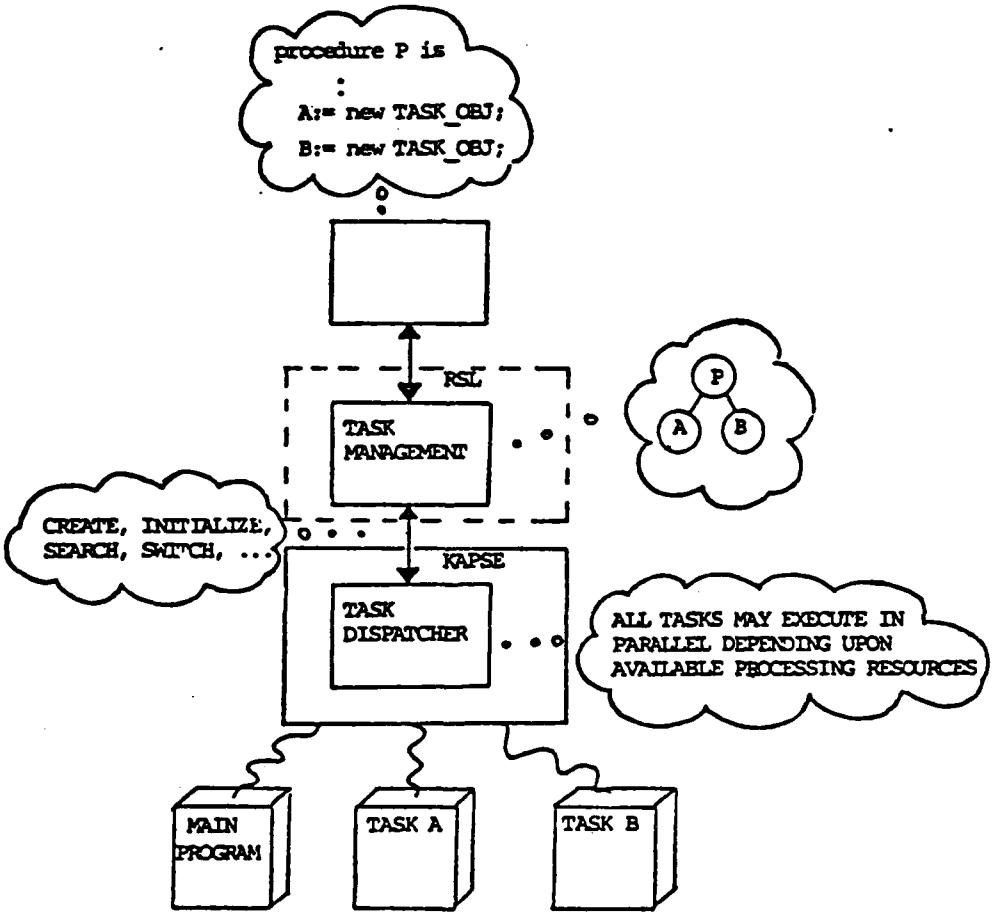


FIGURE-7 : ENCAPSULATION OF LOW LEVEL EXECUTION SERVICES IN THE KAPSE PROMOTES CAPACITY TRANSPARENCY.

The integration of task oriented services into the KAPSE upgrades the code execution service interface, and necessitates the KAPSE assume the role of a general purpose process administrator or manager. This is particularly true when the KAPSE executes upon a "bare machine". A consequence of this role is the provision of efficient synchronization protocols within the KAPSE, if economic and reliable implementations are to be achieved. Also, it may add to the interface, innovative services that facilitate Capacity Transparency. These issues are the subject of current research, and recent work [12] has reported modest advances that may expedite the upgrade in functionality.

The rationale for this discussion has been to present an argument for an APSE-KAPSE interface for unified code execution services that is both Capacity Transparent and semantically complete. This interface is proposed as an essential area for immediate study by the KIT, since it is critical that the execution domain of an APSE program have a precise definition. The principal contribution of this paper has been to suggest that this interface include Ada task dispatching semantics, and that a transportable Task Manager be made available as a part of the Ada runtime system.

Bibliography

[1] Ada Programming Language; Department of Defense; Report MIL-STD-1815; 1980-12-10.

[2] Ada Compiler Validation Implementers' Guide; SofTech, Inc., Report 1067-2.3; 1980-10-01.

[3] Requirements for Ada Programming Support Environments: STONEMAN: 1980-02.

[4] Fisher, DA; Design Issues for Ada Programming Support Environments: A Catalogue of Issues; Science Applications, Inc.; Report SAI-81-289-WA; 1980-10.

[5] Design Evaluation Report for the Ada Integrated Environment; Computer Sciences Corporation and Software Engineering Associates, Inc.; 1982-05-11.

[6] Ada Integrated Environment: Computer Program Development Specification, Part 1; Computer Sciences Corporation and Software Engineering Associates Inc,; 1981-03-15.

[7] Ada Software Environment: Computer Program Development Specification; Texas Instruments, Inc,; 1982-03-15.

[8] Computer Program Development Specification for Ada Integrated Environment: KAPSE/Database Type 85; Intermetrics, Inc. and Massachusetts Computer Associates, Inc.; Report IR-678; 1981-03-13.

[9] Ada Language System; KAPSE B5 Specification; SofTech, Inc.; Report CR-CP-0059-B83; 1981-08.

[10] Ada Language System; VAX/VMS Runtime Support Library B5 Specification; SofTech, Inc.; Report CR-CP-0059-B20; 1981-07.

[11] U.K. Ada Study; Final Technical Report Volume 2; Department of Industry; 1981-06.

[12] Denning, P.J., et al; Low Contention Semaphores and Ready Lists; CACM Volume 24, No. 10; 1981-10.

TOOL PORTABILITY AS A FUNCTION OF APSE ACCEPTANCE

Dr. Steve Glassman
(Teledyne Systems)

The amount and quality of work that has gone into the Ada language program is impressive. We are currently in the midst of a cycle that will see the introduction and diffusion of an initial capability, and have reason to hope that the first reviews from real users will be, in the main, positive and constructive, indicating that the concept of a standard language employed within a somewhat controlled environment is a workable one within its targeted community.

My concern is that we are not working hard enough to ensure good reviews. Specifically, it seems to me that it is time for the technical emphasis to recede somewhat in favor of increased attention to how the language and its environment will be introduced to the user community. I want to use the subject of installation-specific software tools to illustrate my concern, but I hope that those parts of my argument that survive your scrutiny will also be applied to the broader issue first raised.

We can regard the portability aspect of software tools as describing a continuum that moves from an environment of unique and special purpose tools with little or no portability, to one of completely standard, broadly relevant tools with high portability. I think we might agree that neither pole of this continuum accurately describes our expectations for Ada tools. Obviously, we would like to end up as close to the portability pole as possible. Toward this end we hope to close the gap by compromise: we will allow, even,in some cases, encourage the development of installation-specific tools, but demand they be employed within a controlled environment. In this way we hope that new tools will, of necessity, be designed so as to fit within this environment, thus confirming a

degree of portability. We will even furnish an initial tool kit, both for actual use, and to serve as a model for its own extension and for the development of new tools.

This is an attractive idea, but its success hinges on the degree to which the Ada Programming Support Environment (APSE) is accepted by the users. The words user acceptance occur frequently in most of the Ada literature I have read. But I think the issue has, nonetheless, escaped a complete thinking-through.

I am concerned that apparent acceptance based on a willingness to install and evaluate an initial set of tools could rapidly erode if users perceive few incentives for embracing the requirements of APSE conformance.

To see how this could happen, we need to consider the make-up of the user community. It will not change its fundamental character with the introduction of Ada. It is composed primarily of defense contractors of various sizes and capabilities, each of whom has as a major driving function the making of profit. Most have some limited flexibility to involve themselves in projects that do not directly serve that aim. To the extent possible, i.e., profitable, they can be expected to respond favorably to the advent of Ada and its support environments. That is, we hope that by mandating the use of Ada, and by providing an initial set of software development tools and a clear path for their logical and generally standard extension, the community will perceive a cost-beneficial environment within which to pursue business objectives.

History, however, teaches us that contractors respond to contracts. And, although we expect to contractually mandate use of the Ada language, are we also prepared to require the use of a particular programming environment? If not, and there are complex legal, economic, and policy issues raised by doing so, there is a likelihood that users will employ the tools that "come with" Ada, but will make

no continuing internal effort to maintain the integrity of the APSE. For tool development, this implies business as usual: develop and use tools that enhance the effectiveness of customary approaches to software development, ignoring damage done to the APSE concept.

The obvious counter-arguments are, first, that tool development will emerge as a new business opportunity, with many vendors developing tools that do conform to the envisioned APSE evolution path. Thus, the major contractors will have a choice: develop their own tools at cost X, or purchase tools from such a vendor at cost Y. Where cost Y is lower, the APSE concept could benefit to the extent that some number of contractors follow the APSE path; thus, portability would be enhanced. The second argument holds that the contractor community will perceive that strict adherence to APSE conventions can confer a business advantage. In this case the fixed and continuing costs of skewing tool development toward an APSE will be regarded as an investment justified by returns in the form of enhanced business potential.

Far from advocating the superiority of any of the positions outlined above, my claim is exactly that we have no way to judge the matter and, in particular, have no reason to anticipate an outcome particularly favorable to APSE acceptance nor, therefore, to Ada's portability objectives. It seems clear to me that this area has not received the attention it requires, that we must consider incentives that go beyond a set of technical features that will encourage the user community to move in the direction we hope is the right one. With such help, the task will be difficult but achievable. Requiring the use of a language and simply offering what could be a superior environment for its use seems unnecessarily risky.

A short digression will motivate my concern. During the 1960's opinion surfaced that the World Wide Military Command and Control System (WWMCCS) was ineffective as a system because each participating computer installation was

essentially a standalone operation. There was no concept of operations sufficiently well emplaced to permit well coordinated inter-site activities. As a result, each site pursued its own military objectives, implicitly relegating its WWMCCS objectives to second place.

To combat this problem, the concepts of standardization and portability were applied to 35 WWMCCS computer installations. The DoD was committed to standardizing the hardware and much of the software at all sites so that each site had sufficient computational resources to support its unique defense missions at least as well as before and, in addition, should a crisis arise, all relevant sites could by virtue of standard resources and procedures act in concert as one or as several groups of "inter-operable" and intercommunicating teams. The instructive points are:

- Honeywell 6000 series computer equipment was (and is) the hardware base. The GCOS operating system provided one software standard, and there were contractual prohibitions against its modifications for any purpose. Application systems both developmental and fielded were subjected to an oversight mechanism that restricted their programming languages and the directions in which they evolved.
- The DoD and each military service had separate organizations with responsibility for, among other things, coordinating their sometimes competing objectives. These offices acted on the basis of written and unwritten guidelines and were staffed, as a rule, with competent and dedicated individuals.

The WWMCCS environment of the 1970's was one of high expectations with committed and knowledgeable personnel at all levels of a complex organizational hierarchy. Moreover, they had in-hand several reasonably well tested products

upon which to base their standardization hopes: Computer hardware, an operating system, newly modified but not particularly novel, and several large application systems (i.e., large software/data base combinations). There was no question of whether the concept would be accepted, or of its ultimate success. That there were serious problems involved was understood to the best of anyone's ability, but there was great faith that determination and hard work would prevail to produce, if not the fully envisioned product, at least a close approach, and a significant improvement over what was possible before.

Twelve years later the WWMCCS program is minus over $1 Billion, yet is hardly different in capabilities or in potential from the original loose federation of largely unique computer installations. Lest I be accused of over simplification, let us agree that the WWMCCS concept extends far beyond these computer centers, and is a vast and complex beast, many parts of which work well now and will improve as time and men advance. However, the computational subsystem is still a huge entity in its own right, and stands in relation to the WWMCCS concept as a whole much as the Ada language effort stands in relation to the software development industry as a whole.

The WWMCCS objective was to improve the overall effectiveness of command and control by offering improved communications and information processing tools. The tools were to be standard, portable and in effect, if not in name, constituted an attempt at maintaining standard environments for both operations and development--much as we currently envision the results of the Ada program. The WWMCCS effort is widely regarded as a failure, and I believe there are lessons to be learned from that experience that could serve us well at this point in the Ada effort. How did the WWMCCS community standardize so large a part of their technology and still fail to derive the expected benefits?

Quickly summarized, a fatal problem of the WWMCCS effort was poor planning and faulty implementation. The idea was to establish a centralized group to act as a system "mother" for the standard base of hardware and software, and as a clearing house for potential additions or other proposed changes to the computational environment. However appealing the idea, its implementation can be characterized as follows:

- Insufficient technical staff. In some areas the problem is numbers; in others that of competence.
- Poor organizational position. The group's responsibilities overlapped those of its parent organization.
- Fragmented funding. The group did not control its own budget.
- Disinterested management. With few exceptions the group's leaders saw the assignment as one of "marking time".
- Excess responsibilities/inadequate authority. The group was expected to develop and fund an R&D effort in applied computer science in addition to its technical oversight responsibilities. But, without budget control, they had little say in how funds were to be allocated, and could not fight the imposition.
- Lack of a strong organizational focus. With a few exceptions, the whole group lacked a sense of purpose, which discouraged innovative thinking, and encouraged muddling-through as a way of life.

The Ada objective is to improve our war fighting capabilities by offering improved software for systems increasingly dependent on embedded computers. But, although Ada boasts a strong technical base, failing to manage the users' introduction to it seems careless in light of the WWMCCS and similar experiences.

Consider, for example, that a standard programming support environment is an experimental concept, and not without disadvantages for some users. Leaving aside the coercive aspect of requiring use of the language itself, the APSE effort will sink or swim at the level of the individual software engineer. What will motivate him or her to embrace the APSE concept? Several qualities come to mind which, if present, could have the desired effect:

1) Belief that an APSE will make their job easier with acceptable increase in effort.

2) Belief that an APSE will improve their product with acceptable increase in effort.

3) Direct corporate orders to learn, use, and promulgate the entire Ada suite.

I want to ignore number 3) because I believe it unlikely; we may wish to discuss it later. The "acceptable increase in effort" mentioned in 1) and 2) above, gets us back to the issue of software tools. With a careful introduction to its tools, the user might ease into an APSE, be receptive to its benefits, and wish to promote them, e.g., by suggesting enhancements and by developing new tools with an eye toward minimal disruption of a recognizably superior software development environment. In such a scenario, the development of installation-specific tools that cannot be fit to an APSE might be minimized - occurring only in special cases where real project pecularities exist. Under these conditions, the APSE could have the support of its targeted user group, and the experiment as to whether or not a standard HOL and an associated support environment is the key to improved software outcomes will be under way. The verdict should then be based on the merits of the system itself, unclouded by issues of poor implementation, management, and the like.

If, on the other hand, users encounter an initial APSE package with little or no effort to guide implementation, to emphasize its benefits in particular contexts, to anticipate and help to resolve problems, or to provide resources for communicating valuable experiences throughout the user community then, no matter how technically capable, smoothly interfaced, well documented, and configuration controlled, users will resist what can be resisted. In the extreme, as mentioned earlier, they will use the language because they must, but waiver applications will remain common. The environment will languish because other tools are already available - no matter that they do not fit into what is perceived as an unachieved ideal. Portability will go down the drain as tools and environments diverge, and the Ada experiment could fail without ever gathering data. In the end, the language will be just another language, and a major hoped-for benefit -- that of encouraging a software engineering approach to problem solution, with code development a result of that approach instead of an input to the problem -- will not be achieved.

I believe the Ada community should take a more active role to encourage the former scenario. Toward that end I propose first that this group amplify and weigh the points I have briefly outlined. If we can agree that a more thorough approach to APSE introduction is important, then we should prepare a recommendation for the KAPSE Interface Team. Jumping the gun a little, I can see several elements that might be considered. Drawing on the lessons of the past, it is crucial that the Ada Support Facility, as it is described in the STONEMAN document, be in-place and up to speed as regards document availability, easy access to expert consultants, and other resources before these are requested by the earliest general user. The ASF must not begin its operations in catch-up mode; experience shows that it may never pull ahead.

I suggest special efforts go into preparing manuals to guide APSE implementation, and that these, especially, be available prior to requests for the information they hold. I would also suggest the periodic reports on status and planning mentioned on STONEMAN include, for example, a regularly updated newsletter describing existing tools, tools currently under consideration as standards, and tools in various stages of development.

A final example is to ensure full benefits from the user groups to be supported by the ASF. The useful information exchange that is the object of such groups has often been diluted by careless implementation. User groups should bring together, on a regularly scheduled basis, those individuals actually involved in the development and use of e.g., software tools; not their managers. Moreover, these groups have a dual utility. Besides the hoped for cross-fertilization of users from different locales and market interests, such groups represent a window on APSE employment in the different sectors. Their meetings are, thus, an invaluable source of feedback for those concerned with Ada's success and future growth -- provided there are resources for gathering and processing such information.

What users experience in the comparatively short period during which Ada and its APSE are introduced will play a major role in determining the success of the Ada effort. Successful introduction of the APSE concept requires early and careful planning. The result should be widespread user acceptance of a superior scheme, and a growing body of participants who are self-motivated to adhere to the conventions of a standard environment. This means increased portability, not only for tools, but for resources and products in general. At that point in time the Ada experiment will begin in earnest.

TRANSPORTABILITY ISSUES

Eric Griesheimer
(McDonnel Douglas)

Introduction

"A ... goal of great importance in some areas of Ada usage, such as within the DoD, is that of portability both of user programs and of the software tools within the APSE." A possible approach to the transportability problem is the subject of this paper.

The Mover Tool

To accomplish the transport of any database object(s) to another APSE environment, let's postulate a "Mover" tool. By assumption, the Mover can transport any Ada object to another system via an appropriate carrier.

The carrier may consist of physical media, or it may be electronic. In either case there is a requirement for hardware to support the carrier function. There is also a need for corresponding KAPSE functions to drive the hardware.

Carrier Alternatives

Program and tool migration from one APSE to another requires that the APSEs be hardware and software carrier compatible. This can be achieved most easily by adoption of carrier standards. For example:

Physical Media

Designation of the most commonly used magnetic media (tapes and floppy disks, for instance) would make a big step toward compatibility if other physical attributes were also specified - density, recording mode, and blocking factor, for example.

Electronic Transfer

The obvious candidates here are the well-known high speed packet switching networks. There should be an option, however, to choose a relatively slow and inexpensive (but reliable) method of transport. Typical hardware would employ standard serial terminal ports driving the dial-up networks via modems.

Carrier Protocol

Standard redundancy techniques (parity, checksums - even repeated transmission, perhaps) should be utilized for either media or electronic transfer.

Electronic link protocol should provide for automatic retransmission as necessary. All carrier protocol is envisioned as a KAPSE function, although that may not always be necessary.

The Content Question

Ideally, the Mover should be able to transfer any object in the APSE database, subject to access restrictions. The content issue is considered carrier independent.

Added Attributes

When a user program or APSE tool is brought to a new APSE, the history part of its object attributes should be updated to show its origin, time of transfer, and identity of the agents of transfer - i.e., the Mover and its KAPSE.

The Mover could attach transport history information to the subject object as a transport configuration object. The transport history could then be removed or retained at the receiving APSE, as desired.

The Bilateral Mover

The Mover should be designed to perform at user request either the sending or receiving transport function. For electronic links the transport function could be initiated from either end.

A KAPSE Link Primitive

It is suggested that every KASPE be furnished with a low level link primitive to support error free communication via asynchronous lines. Then, by simply adding a modem to any user terminal port, a remote APSE host could establish low bandwidth error free communications with any other, including a generalized object transport capability.

THE EFFECT OF DATA DESIGN ON APSE DATA AND TOOL PORTABILITY

Judith S. Kerner and Steven D. Litvintchouk
(Norden Systems)

Abstract

Problems that may inhibit transportability of data and tools between the Air Force Ada Integrated Environment (AIE) and the Army Ada Language System (ALS) result from differences in the structures of the data bases they provide. The data structures, as "viewed" by their users, are incompatible. The relationships between the data items in a data base are defined by implicit information that is unique to that data base; the implicit information is absent from or inconsistent with the other data base. Therefore, a tool for transporting data structures from one data base (the "source") to the other data base (the "target") may not have access to the information required to generate the data structures in the target data base. Further, since this tool (and others) may depend upon the implicit information peculiar to each data base design, the tools may not work on other Minimal Ada Programming Support Environments (MAPSEs) which provide dissimilar views of the data.

Text

Problems of transportability of data and tools between the Air Force Ada Integrated Environment (AIE) and Army Ada Language System (ALS) result from differences in the designs of the data bases they provide.

The overall structures of the data base designs are similar in that each provides a host-independent, basically hierarchical (tree-structured), view of the data to the users of the APSE. However, there are significant differences in detail. Among these differences are:

1. . Stored objects. The AIE provides for three different basic classes of data base objects. Simple objects resemble flat files. Window objects are references to other parts of the database (i.e. other "branches" of the tree structure). Composite objects are sets of objects, from these three classes. These composite objects, therefore, provide the user with an n-dimensional organization for data items; in addition, the AIE MAPSE provides facilities for accessing slices ("partitions") of these n-dimensional structures.

The ALS data base management system provides three different classes of objects as well; unfortunately, these are not the same as those provided by the AIE data base. Files are simply flat files. Directory nodes are used to group other nodes (acting as a "logical AND" of the nodes making up the directory). Variation headers are used to support collections of related objects which comprise "equal" alternatives (acting as a "logical OR" of the nodes making up the variation set).

(The AIE also provides some special kinds of objects, some of which have no analogue in the ALS. Private objects, for example, provide the ALS user with encapsulated abstract data objects analogous to Ada private types.)

2. Access to other portions of the database. This capability is provided in the AIE through window objects, mentioned above. In the ALS, this capability is provided through associations. The associations of a node are collections of pointers to other nodes. (Note that the access restrictions imposed by the APSEs are also different; the access rights of an AIE window object cannot exceed those of the parent of the object; the ALS apparently does not provide a similar restriction.)

3. Attributes. The ALS data base management system provides a much larger number of predefined attributes to the user than does the AIE. (Refer to the Softech Ada Language System Specification, CR-CP-0059-A00, section 50.7, for a comprehensive list.) Even predefined attributes provided by both systems, however, tend to differ in detail. For example, the AIE history attribute appears to provide somewhat more information than does the ALS history attribute.

4. "Canonical" data base. An AIE data base has the following four main subtrees extending from the root: SYSTEM, USERS, TOOLS, PROJECTS. The ALS data base has the following three main subtrees extending from its root: ALS, MAPSE, USERS.

5. Synchronization control. To provide synchronized access to portions of the data base, the AIE provides the following "reserve modes" to the user: EXCLUSIVE_WRITE, READ_ONLY, SNAPSHOT_READ, SHARED_STREAM, SHARED_RANDOM. (Refer to the AIE Contractor AIE KAPSE/Database B5 Specification, IR-678, section 3.2.5.1, for a full explanation of these terms.) The ALS apparently provides no similar capabilities.

The data structures as "viewed" by the users of the different data base designs are therefore incompatible; each data base provides certain implicit information about the relationships between the data items which is not provided by the other data base.

This makes it difficult to design a general tool for transporting data structures from any one data base (the "source") to any other data base (the "target"); the implicit information about the way the data should be structured in the target data base is totally specific to the target, meaning that a specific transporting tool would have to be written for each target data base to which data is to be exported.

Even if information about the target data base is available to the transporting tool, it should be apparent that the only data structures which can be successfully transported from the source data base to the target are those which make use only of those features and capabilities common to the two data bases. Obviously, if the target data base contains a feature which the source data base does not, no data structure exported from the source data base will contain such a feature; if the source data base contains a feature which the target data base does not, no tool running on the target data base will know what to do with such a feature if it is incorporated in the data structures being transported.

But if we impose on tool designers the constraint that they assume only the existence of features common to both APSE data base designs in the tools they create, the capabilities of the tools will necessarily be quite limited, since the two APSE data base systems have so few features in common. Further, as more and more APSE's are designed by more and more vendors, the number of features they all have in common will tend toward zero, making it increasingly difficult (if not impossible) to design portable tools for data base manipulation.

Bear in mind, however, that the view of the data provided to the user by a data base is in general dependent both on the basic structure of the data base (provided by the data base management system; i.e. APSE), and on the tools designed to access the data base. (For example, it is possible to implement a fully relational data base on a host data management system providing only flat files, by designing sufficiently powerful programs for accessing the flat files in the required ways.) Therefore, "data base portability" should involve transporting both the data itself, and the tools for accessing the data.

Unfortunately, such tools are themselves data objects (presumably Ada source code program files); transporting a library of such tools from one data base to another may well cause the same problem as transporting any other data structures: namely, mapping the structure of the source library onto the target library. Perhaps these problems can be overcome by a sort of bootstrapping process, in which a simple tool (which depends as little as possible upon the structure of the source libraries and data base) is utilized to transport the set of tools which do depend on these structures (one such tool being the one which transports the rest of the data base); finally, the transporting tool is invoked to transport the rest of the data base.

Even if the tools can be transported successfully, however, they may not work correctly on the target data base. A tool currently running on a source data base may well make use of features peculiar to that source data base. Such features may not exist in the target data base at all. If the features are provided through various packages, data types, etc., in the source data base, the tool source code may not compile successfully in the target data base, since these packages and data types may not be defined in that data base.

Even if the names are defined in the target data base, we are only guaranteed that their syntax and static semantics are the same as in the source data base; the feature may still have a different dynamic semantics. (For example, both data bases may define the history attribute data type to be a series of text lines recording the derivation of the object; but the formats of the texts in the two data bases might still be different.) In this case, the tool may compile and link successfully, but not execute properly.

In order for such a tool to be able to work on a target data base, therefore, some sort of interface between the tool and the target data base could be designed which would emulate the operation of the source data base in the target APSE. It might be desirable, in fact, to design a portable interface between all tools and the underlying APSE, providing a set of standard data base accessing capabilities which is judged adequate for supporting software development on an APSE. This would essentially standardize the view of the data provided the user by any APSE.

However, efficiency will still suffer from this method of transportability, since a tool which is transported from a source data base is not making full use of sophisticated features which the target may provide. For example, suppose a tool exists which provides the user of the ALS data base system with n-dimensional composite objects similar to those provided by the AIE. Since such data organization is not directly available in the ALS, the tool must implement it on the underlying tree structures provided by the ALS. If the tool is then transported to the AIE, it will probably not work as efficiently as a tool designed to make maximum use of the AIE's capabilities for manipulating such data objects. If achieving maximum efficiency is important, then both the tools and data structure design provided on the source data base will have to be redesigned to provide the same features on the target data base (as specified by the original requirements specification), but perhaps through a different design--providing certain features in the target data structures that used to be provided by the source tool, or providing certain features in the target tool that used to be provided by the source data structures, or both.

In conclusion, differences in the designs of the data bases provided by different APSEs present obstacles to the development of efficient general techniques for transporting data between data bases. In addition, it appears likely that transporting the user's entire view of the data will involve transporting as well all tools which access the data; this presents still other problems.

Each of these problems must be solved in order to provide full transportability of data between any two APSEs.

DATABASE PORTABILITY ISSUES IN THE KAPSE

Reed S. Kotler
(Lockheed Missiles and Space)

In this paper I propose a set of minimal requirements to assure KAPSE database portability.

In examing issues of KAPSE related portability, one is confronted with the problem of transferring database objects from one KAPSE to another. Stoneman requires of the KAPSE database, in section 2.B.12, that: "The declarations which are made visible by the KAPSE are given in one or more Ada package specifications. These specifications will include declarations of the primitive operations that are available to any tool in the APSE." These specifications, while given in Ada, present two basic problems:

1) Ada compilers for different machines may have different pragmas, predefined language attributes, representation specifications and STANDARD package. This may result in non-isomorphic mappings between different database representations as a result.
2) Already there exist two distinct KAPSE designs, one for the Army Ada Language System (1) and one for the Air Force Ada Integrated Environment (2). The momentum already building behind these efforts, as well as efforts by computer manufacturers, make it unlikely that a single KAPSE database specification will emerge.

KAPSE database objects may be those which fall under the class of standardized objects, (such as DIANA), or may be of some user or project defined types. In transferring database objects from one KAPSE to another, (in the simplest view), the database objects will be transformed to some kind of external form, dumped to tape and reloaded and transformed to the internal form of the new KAPSE. This type of transformation process has already been addressed for database objects of type DIANA (3).

It is the position of this paper that:

1) The external form for a given KAPSE database object be an Ada program capable of generating the database object,
2) The Ada program thus created is moved to the new KAPSE, compiled and run, thus creating the database object on the new KAPSE.
3) When reading/writing from/to database objects from an Ada program, the association between the Ada objects being read/written and the definition of the corresponding Ada object types must be maintained.
4) A tool, which I will call the KAPSE Interface Description Processor (KIDP), must be provided. The KIDP must be capable of automatically generating both the external form of a given database object, as described above, and a set of primitive Ada subprograms for reading/writing Ada objects from/to a database object. The KIDP takes as input Ada context specifications. These context specifications define the domain of Ada object types which can be written to a given database object type.

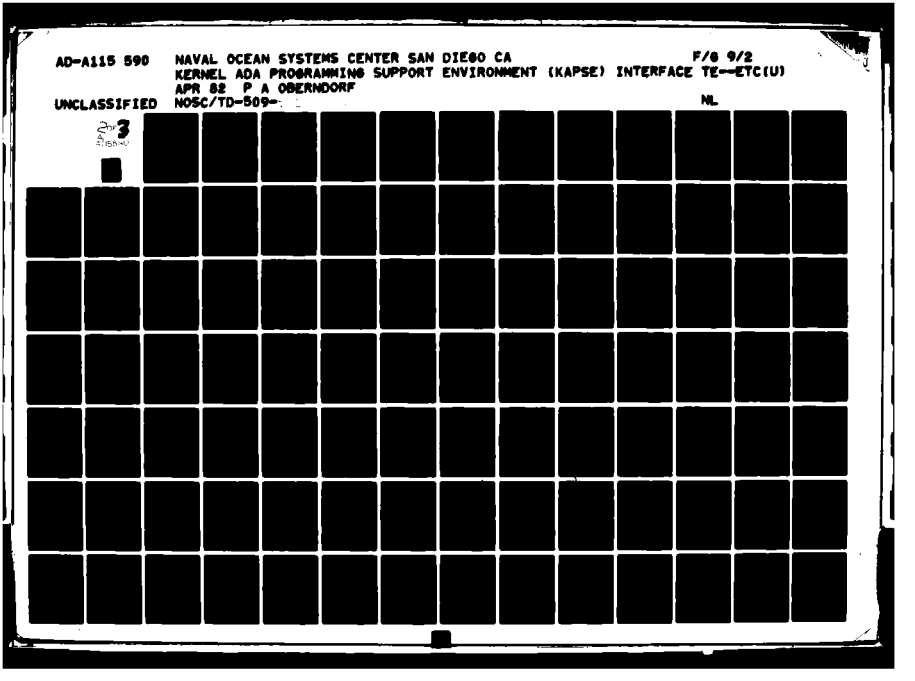
The process of standardizing a KAPSE requires that the given KAPSE must support the functionality described above. Additionally, one may submit a set of concrete representation extensions to the input of the KIDP, to force various Ada object types to be represented in some specific manner in the KAPSE database.

Basic Philosophy

We begin this discussion by analyzing the database situation depicted in Figure 1. In Figure 1 the following graphical conventions are observed:

1. Distinct database objects are enclosed in solid boxes. These are in turn numbered just above the upper right hand corner for reference.

2. Divisions within a database object are indicated with dashed lines without arrows.


AD-A115 590
NAVAL OCEAN SYSTEMS CENTER SAN DIEGO CA
F/G 9/2
KERNEL ADA PROGRAMMING SUPPORT ENVIRONMENT (KAPSE) INTERFACE TE--ETC(U)
APR 82 P A OBERNDORF
UNCLASSIFIED
NOSC/TD-509-
NL
3

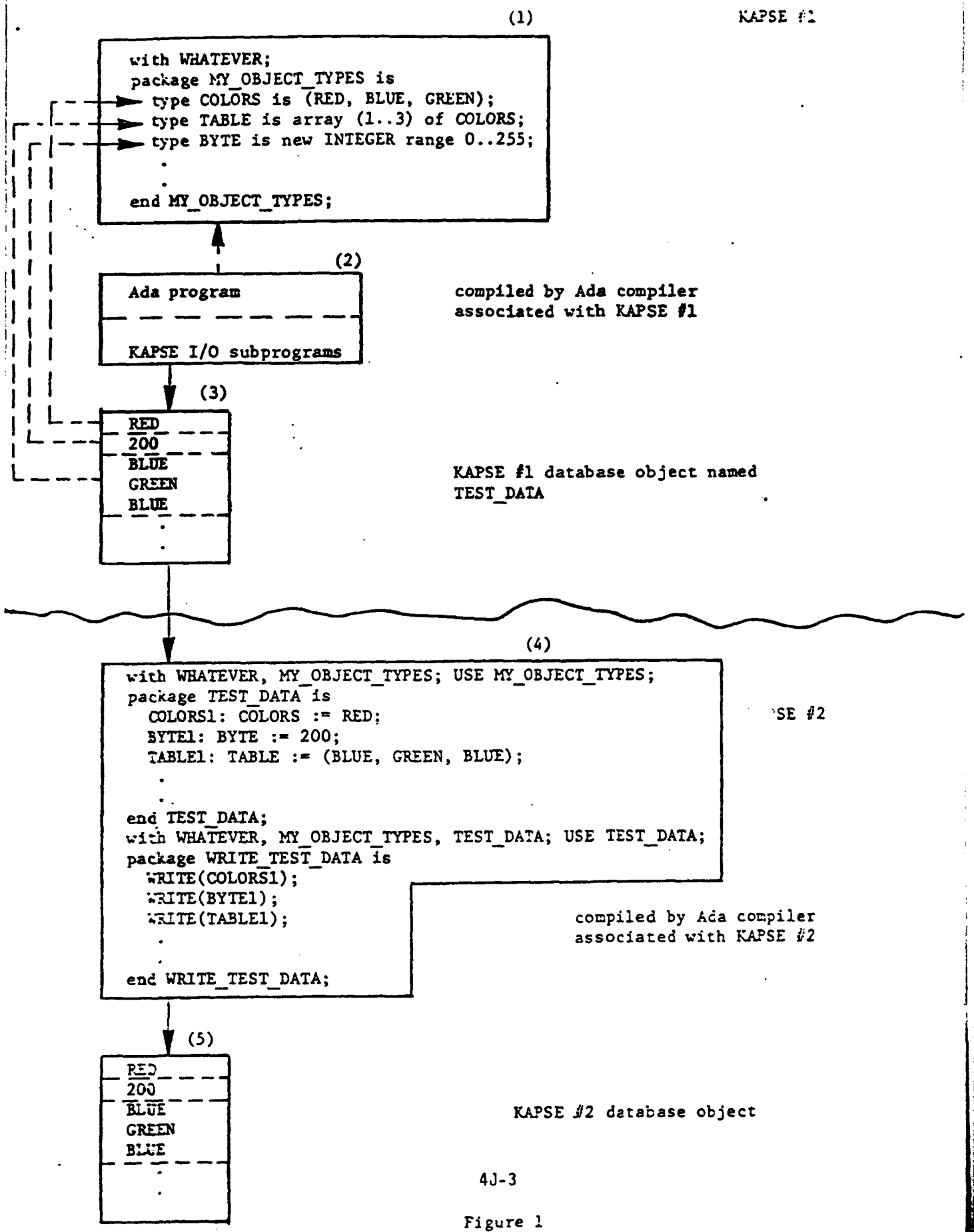

(1)
KAPSE #1
with WHATEVER;
package MY_OBJECT_TYPES is
type COLORS is (RED, BLUE, GREEN);
type TABLE is array (1..3) of COLORS;
type BYTE is new INTEGER range 0..255;
end MY_OBJECT_TYPES;
(2)
Ada program
KAPSE I/O subprograms
compiled by Ada compiler
associated with KAPSE #1
(3)
RED
200
BLUE
GREEN
BLUE
KAPSE #1 database object named
TEST_DATA
(4)
with WHATEVER, MY_OBJECT_TYPES; USE MY_OBJECT_TYPES;
package TEST_DATA is
COLORS1: COLORS := RED;
BYTE1: BYTE := 200;
TABLE1: TABLE := (BLUE, GREEN, BLUE);
end TEST_DATA;
with WHATEVER, MY_OBJECT_TYPES, TEST_DATA; USE TEST_DATA;
package WRITE_TEST_DATA is
WRITE(COLORS1);
WRITE(BYTE1);
WRITE(TABLE1);
end WRITE_TEST_DATA;
SE #2
compiled by Ada compiler
associated with KAPSE #2
(5)
RED
200
BLUE
GREEN
BLUE
KAPSE #2 database object

Figure 1

3. Database objects which are used to create new database objects are connected to the new database object by solid lines with arrows.

4. Associations are indicated with dashed lines with arrows.

KAPSE database objects will hereafter be referred to as KD objects and Ada objects simply as objects.

We now continue with a step-by-step discussion of the contents of each solid box in Figure 1.

Box # 1: This serves as the primary abstract description of the object types of objects which make up the KD object called TEST_DATA.

Box # 2: This is an Ada program which contains some user written code and some code automatically generated by the KIDP. Its input context specification is assumed to be:

```
with MY_OBJECT_TYPES;
```

The automatically generated code consists of read/write subprograms for any objects which could be defined using the package MY_OBJECT_TYPES;

Box # 3: This is a KD object created by the program of Box #2. The exact representation of the KD object is unimportant for purposes of database portability. What is important is that the associations between the Ada package definition(s) used to define the types of the objects and the objects themselves is preserved. Most likely the associative information should be kept separate from the actual KD object in the database so as not to violate Stoneman requirement 5.2.A which states that "the database shall not impose restrictions on the format of information stored in the object" (KD object).

Box # 4: This is an external representation for the KD object in Box #3. Each object in Box #3 has been assigned an automatically generated name, declared to be of object type corresponding to the object in Box #3, and initialized according to the object's value in Box #3.

When the KD object of Box #4 is compiled and run in a new KAPSE-Ada compiler configuration, then the KD object will have been effectively transported.

A few notes are required here. The external representation need not actually be Ada source. Since it is automatically generated by the KIDP, DIANA,or perhaps some form of compacted DIANA, would suffice to accomplish a more space efficient implementation.

As a bonus, by using Ada/DIANA source as the external representation, a number of immediate benefits arise which are unrelated to portability:

a) The KD object in external form can easily be understood by anyone possessing a knowledge of Ada. If DIANA is used, a source reconstruction tool can be used to create an Ada source version of it.

b) Tools used to perform various types of static analyse for Ada programs can be used to analyze KD objects.

c) KD object editing can be accomplished by first transforming the KD object to external Ada/DIANA representation, then editing it with a text editor, and lastly compiling and running the edited program which is the external Ada/DIANA source representation of the KD object.

An important point to make is that we have implicitly assumed that the entire KD object could be loaded into memory at once. In some KAPSE's this would not be possible. In such cases an Ada interpreter using random access mass storage devices for virtual memory would have to be provided. Note, however, that the interpreter would only have to be able to interpret a small amount of the Ada language.

Box #5: This is the KD object after being transported to a new KAPSE. This KD object contains the same information as the KD object of Box #3. However, the exact representation may be completely different.

Specification of Abstract Structure for a Given KD Object

As demonstrated in Section I, specification of the abstract structure for a given KD object involves providing the KIDP with the necessary Ada context specifications needed to define objects that are read/written from/to a given KD object.

The only restrictions placed on the allowable object types are the following:

1) Objects which are to be read/written must be declared to be of a type for which assignment is possible. This is because the reading of an object involves assignment.--this causes objects of limited private type to be excluded.

2) Representation specifications and related pragmas in the context specifications are ignored. This may not be a necessary condition; however at the present time I have not had enough time to analyze the technical ramifications concerning this.

Creation of Primitive Read/Write KAPSE Subprograms

For each object type contained in a given abstract KD object description, a set of read/write subprograms is produced by the KIDP. The basic semantics of these subprograms is as follows:

1) Only objects not possessing access types are read/written in a straight-forward manner.

2) In the case of objects with access types, all objects contained in a transitive closure of the access relation are read/written. In other words, if A is a record component which is an access for B which in turn accesses C, then A,B and C are all read/

written. Because of the strong typing in Ada, the code to do the access chasing can be generated by a number of fairly straight-forward algorithms. When written, all access objects contain a virtual address within the KD object. When read, virtual access address is converted to a memory address.

The remainder of this section is devoted to issues not relevant to portability but which are relevant to certain implications of the above semantics for access type reading/writing on KAPSE database efficiency.

A straight-forward implementation of the access method described above could require considerable overhead when dealing with large KD objects. This is because in order to read/write KD objects with access types one would conceivably have to read/write the entire KD object as opposed to selectively reading/writing what is needed.

To avoid this problem, we may implement a lazy evaluation scheme for reading access objects. In lazy evaluation, upon encountering a read of an access object we would not actually read all objects contained in a transitive closure of the access relation. Instead we would only read the single access object which we know contains a virtual access to the KD object we read from. Upon encountering a dereference of the access object, we would read the required object into memeory and then change the virtual access of the original access object to a memory access. This scheme would required some coordination between the Ada compiler and associated runtime system.

Lazy evaluation schemes for writing to KD objects which are opened in an update mode can also be implemented. These schemes are more complex than those for lazy reading. A discussion of these would be too lengthy for this paper. It needs to be reiterated, though, that these schemes do not have a different semantics from the straight-forward implementation, only a different implementation which is much more efficient.

External Representation of KD Objects

As in Section I, a package similar to the one contained in Box #4 is created, with each object assigned an automatically generated name, declared to correspond to the original object type and initialized with value corresponding to those of the original values in the KD object. All access types must be linked together in the corresponding package body as shown in Figure 2.

Next, a procedure for writing the data is created, as in Box #4 of Section I, with each object written in the correct order corresponding to the original KD object.

Concrete Representation Extensions to KIDP Input for a Given KAPSE

Concrete representation specifications allow one to achieve more efficient implementations on a given KAPSE, though they are optional for the standardization effort.

For example, special storing of sparse matrices could be allowed by writing:

```
for MATRIX use SPARSE_REPRESENTATION;
```

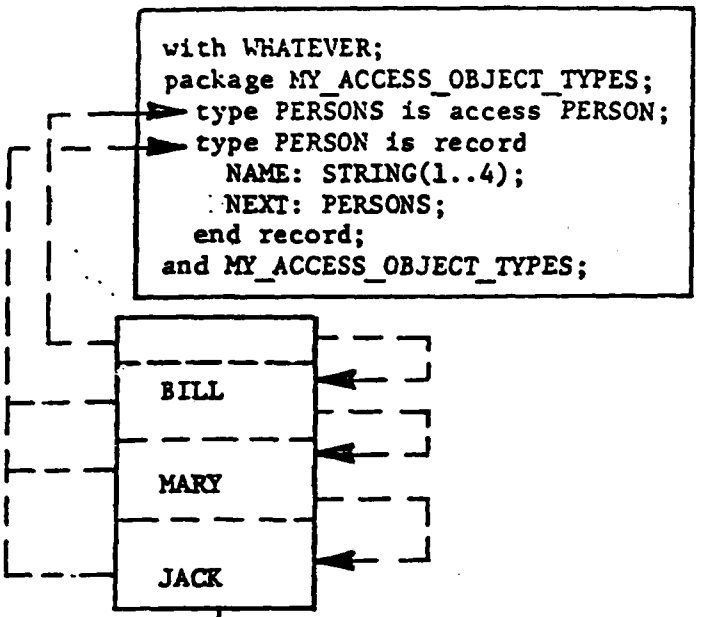
This would cause the array structure to be represented as a linked list with repetitive elements compressed.

Conclusions

In this paper I have presented a minimal set of requirements for KAPSE database standardization which will eliminate potential portability problems.

By using Ada/DIANA source as the external representation of KD objects, we reduce questions of KAPSE database portability to those of Ada program portability. In other words if the Ada programs which use the KD objects can be ported to a new KAPSE, then so can the corresponding KD objects.

By defining a complete set of semantics for reading/writing of Ada objects from/to KD objects, we eliminate the need for a plethora of special purpose methodologies at the KAPSE level. In particular, by the proposed scheme for




```
with WHATEVER, MY_ACCESS_OBJECT_TYPES; USE MY_ACCESS_OBJECT_TYPES;
package TEST_DATA is
  PERSONS1: PERSONS := NULL;
  PERSONS2: PERSONS := new PERSON ("BILL", NULL);
  PERSONS3: PERSONS := new PERSON ("MARY", NULL);
  PERSONS4; PERSONS := new PERSON ("JACK", NULL);
end TEST_DATA;
package body TEST_DATA is
  begin
    PERSONS1 := PERSONS2;
    PERSONS2.NEXT := PERSONS3;
    PERSONS3.NEXT := PERSONS4;
  end TEST DATA;
with WHATEVER, MY_ACCESS_OBJECT_TYPES, TEST_DATA; USE TEST_DATA;
procedure WRITE_TEST_DATA is
  WRITE(PERSONS1);
end WRITE_TEST_DATA;
```

Figure 2

handling of access objects, we eliminate the need for programmers to treat access types in KD objects and those in Ada programs differently. This view of KD objects is that they are merely convenient places to save data structures created by Ada programs. When the data structures are needed again they are brought back into memory in a simple fashion. In other words read and write are inverse one to one mappings from Ada program data structure to KD objects.

The motivation for including the above read/write semantics in this paper on portability is that they are needed to support Ada/DIANA source as the external representation for KD objects. Such a read/write semantics is not limited in usefullness to such support.

There is much work needed to expand the approach of this paper into a working document. I recommend that such an effort be undertaken in conjunction with the KAPSE standardization effort.

Acknowledgements

Many thanks go to Lee Blaine, Dick Wright and Bill Morrison for providing many helpful criticisms and suggestions during the development of this paper. Also thanks to Anita Morris for her typing of this manuscript.

References

1. Ada Language System Specification; Softech Inc., 1981

2. Ada System Specification for Integrated Environment; Intermetrics Inc., 1981

3. DIANA Reference Manual; G. Goos and Wm. A. Wulf, Computer Science Dept., Carnegie-Mellon University, 1981

STANDARDS FOR THE KERNEL ADA PROGRAMMING SUPPORT ENVIRONMENT

J. Eli Lamb
(Bell Laboratory)

Introduction

If Ada and its programming support environment are to supplant existing languages and software engineering environments without enormous cost, they must allow the retention and exploitation of existing proven technology. If the hope that Ada will provide a single language and a uniform environment for software development becomes a tyranny of repression, the hope and the language will die.

An essential goal of Ada and APSE is to reduce the software life-cycle cost. Using an existing technology base for developing new tools has been shown to be both effective and efficient. Implementing Ada support environments on top of proven hardware and operating systems should therefore be a fast and reliable development strategy. KAPSE interface standardization is both necessary and sufficient to encourage this approach.

Alternate approaches include no KAPSE interface standards and complete APSE standardization, spawning separate problems. If no KAPSE standards are established, the goals of portability and cost reduction will not be met; completely incompatible KAPSEs will proliferate. If standards are established and enforced not only for the KAPSE but for the MAPSE and UAPSE, competition will be eliminated and the APSE will have little continuing innovation, leading to stagnation and disuse.

KAPSE standards that both encourage and exploit the retention of existing software technology while providing a uniform, flexible interface to user programs are critical to the success of Ada. Standards for elements of the MAPSE and UAPSE such as the command language interpreter syntax will, however, be both harmful and ignored.

While rejecting explicit interface standards for levels other than the KAPSE, this position relies on KAPSE interface standards, intermediate language specifications, and abstract syntax definitions of compilation units, libraries and executing programs to create an environment in

which conventions will impose order on how MAPSE tools interface with user programs. Further, it is expected that a basic set of MAPSE tools will come into common use through an evolutionary process.

Motivating APSE Development

If it is to succeed, the task of developing a software development environment portable to a variety of hardware and operating systems must be supported enthusiastically by private enterprise. That enthusiastic support can be won only through providing opportunities for profit and individual contribution.

These opportunities can be provided by not discouraging the exploitation of existing technology and products. The standards and requirements for the APSE must enforce a basic functional consistency while allowing and even encouraging innovation and variation in implementation.

While providing opportunities for the development of different solutions to a common problem, we must be certain that there exists no motivation for development of fundamentally incompatible solutions. This balance between variation and standards is, of course, a difficult one. Bell Laboratories believes that the UNIX* system is one that has been successful in achieving this balance. An examination of UNIX and why it has been successful teaches some valuable lessons.

UNIX and Innovation

UNIX encourages innovation by making it inexpensive. How does it do this?

First, the UNIX philosophy apes that of Nature in using a small number of different building blocks to construct larger, more complex entities. UNIX does this by carefully defining a few simple interfaces between each of the elementary building blocks and then allowing them to combine in a variety of ways. Lesson: develop a system with a few simple interfaces.

A second aspect of UNIX is tangential to the first. Each element of UNIX, each program, each process can make a few safe and useful assumptions. Two of the most useful of these assumptions concern the parameter list and the input-output streams. Each process is provided with access to the number of parameters and pointers to them. The count is

---

* UNIX is a trademark of Bell Laboratories.

always an integer and the arguments are null-terminated strings. This simple but guaranteed information greatly reduces the cost of developing and maintaining tools. Likewise, each process is guaranteed three streams for I/O - an input stream, an output stream, and a stream for error messages. Again, these simple guarantees reduce development and maintenance costs while encouraging convention standards. These assumptions together determine much of the character of UNIX software. **Lesson:** guarantee a few fundamental facts about the environment.

UNIX makes innovation inexpensive through separating implementation from interface. A prime example of this is the **stdio** package that has effectively concealed radically different implementations from the user program. The Ada language should help encourage this separation in the APSE. **Lesson:** separate implementation from interface.

A final encouragement to the development of new and different tools is the modularity of the UNIX system. The addition of a new piece of hardware is done easily by writing an entirely separate device driver. This driver looks to all other programs just like any other driver and requires no changes to either the kernel or commands. **Lesson:** encourage modularity.

UNIX and Incompability

How then does UNIX discourage the development of fundamentally incompatible tools? The answer to this question is the same as the answer to the previous one. UNIX discourages fundamental incompatibility by making compatibility so attractive and inexpensive. It simply is not economically justifiable or advantageous to develop a UNIX tool that does not conform to UNIX conventions or does not exploit the UNIX assumptions or services.

Some Recommendations

We recommend serious consideration of the lessons learned from the UNIX system.

We don't believe that the DoD or any other body will be able to enforce APSE standards. The motivation for observing standards must in this context be economy and elegance. We must design the fundamental interfaces and assumptions of the KAPSE to make them inexpensive and attractive to developers of proprietary software. Some cases we will be unable to cover. In those cases where there is some

overriding motivation (extremely awkward architecture, rigid performance requirements, etc.), KAPSE standards will be violated. We must expect this and plan for it.

As an initial approach, we recommend modeling a set of machines and systems to explore how successful proposed interfaces would prove in those environments. Final proposals should be presented relative to these models. Clearly, machines or systems that deviate significantly from the models used will be less able to support a standard KAPSE.

KERNEL INTERFACE REQUIREMENTS BASED ON USER NEEDS

Dr. T. E. Lindquist
(Virginia Institute of Technology)

Abstract

The Naval Ocean Systems Center (NOSC) has been charged with leading the development of a set of conventions and standards that assure the transportability of Ada Programming Support Environment (APSE) tools and data bases. Because of several factors including architecture, host/target configuration, level of tools required, and support environment user training level, the transportability requirements for the KAPSE interface must be carefully constructed. In this paper I discuss how the KAPSE interface is affected by different approaches to the interaction between APSE tools and Ada language users. A hierarchical approach to the KAPSE interface is discussed as a way to satisfy the range of requirements implied by the different approaches to interaction. The necessary relationships between a compilation unit, an intermediate language representation of the unit, and its computation are discussed for an interactive program construction tool. The final section of the paper describes related human-computer interaction studies that are being conducted by the author.

Introduction

Ada represents an interesting change in language processors by binding many aspects of the operating environment with the language itself. Tools such as editors, linkers, libraries, and testing aids are actually specific to the language. This approach has the advantage that tools can be tuned to the Ada language and that writers of Ada software can easily move from system to system. Two pragmatic concerns, however, arise with language specific tools. Firstly, advances in programming support tools, especially for interactive systems, appear frequently through updated versions. Although one might expect that such advances would be incorporated into APSE tools, clearly extra changes might be required at a low level to support these enhancements. The other concern is that an APSE must be flexible enough to accommodate tool needs imposed by various host/target configurations and needs imposed by vastly different user classes. Many have realized and documented the affect that various physical host configurations will have on APSE tools, but additionally, user based considerations will also effect the kernel interface. For instance, in situations where an Ada user is required to work regularly with both an Ada language system and another language system, two vastly different command structures would hinder productivity.

The Naval Ocean Systems Center (NOSC) has been charged with leading the development of a set of conventions and standards that assure the transportability of Ada Programming Support Environments (APSE) tools and data bases. A kernel set of data types and primitive functions, called KAPSE, will be developed and used to implement APSE tools. KAPSE provides the machine dependent interface between the Ada language system and the underlying host. And, through that interface allows APSE tools to be moved from one KAPSE to another. In light of the concerns mentioned above, the problem appears to be to design a KAPSE interface that can serve APSEs with differing host and user requirements and that can change in response to future tool enhancements.

One way that host and user requirements affect the KAPSE interface is through the interactive capabilities of the APSE tools. User needs may dictate that a high degree of interactiveness be present in the APSE. In the section that follows we take a closer look at what these capabilities might be in the context of the APSE command structure and program construction facilities. The intent is not to suggest specific APSE tools, but instead to uncover the implications that similar tools might have on the form of the KAPSE interface. After discussing a highly interactive set of program construction tools, we sketch a suitable KAPSE level computation model to support the tools. The focus of this paper is on interactive issues because they are one topic being covered on a research project with which the author is involved. In the last section relevant studies that are being conducted on the project are briefly discussed.

A KAPSE That Supports Highly Interactive APSE Tools

It is certainly true that more and more software is being developed on what are commonly called interactive systems. Generally, interactive systems are characterized as having a terminal with which the user communicates with a seemingly individualized computer system or language processor. It is important to emphasize that there is a spectrum of interactiveness that extends beyond a two-valued property. The level of interactiveness that a specific tool possesses can be characterized by the atomic units with which the user communicates to the system. For example, a batch processing system is one in which the unit of communication is a deck of cards. Such a system has a low level of interactiveness.

Command Structure. The very basic question of command structure has to do with the format used for command entry. Schneiderman and others have treated this question from a human-factors point of view with fairly conclusive results. The three formats that seem to be most widely used are menu driven, fill-in-the-blanks, and parametric input. Menu driven commands are those in which all the alternatives are placed on the terminal and a choice is requested from the user. Fill-in-the-blanks on the other hand prompts the user for each parameter necessary to complete a command. Generally, menu and fill-in-the-blanks are integrated together whereby the menu is used to select a command and fill-in-the-blanks is used to supply the arguments. Parametric input places the entire burden on the user. The name of the command along with any appropriate parameters are all entered, often on the same line, separated by delimiters.

Menu and fill-in-the-blanks formats are generally used most productively in situations in which users are always learning the system. Such situations arise when the tool is used infrequently or there is a fast turnover of users. Parametric input is often the desired choice for experienced users of a system simply because they get tired of waiting on the other formats.

Certainly, APSE tools such as syntax directed editors fall in the category of fill-in-the-blank input format. The specifics of a language statement are entered in response to a user prompt with these editors. For example, after entering the appropriate code for an if-then-else statement, prompts would be placed on the screen to request the boolean condition, THEN statement, and ELSE statement. One would expect that such tools might be more appropriate for programmers learning Ada than for the experienced user. With regard to APSE's, the most desirable situation is one in which users may tailor the input format to their individual needs. For one user familiar with other APSE tools but rusty in Ada, it might be beneficial to allow syntax directed editing of source along with parametric invoking of all other APSE tools.

Command Processing. Although the different command formats can be seen as providing distinct levels of interactiveness, it is cleaner to classify the interactiveness of commands by the way they are processed. In doing this we will look at three interactive levels of processing.

The lowest and most common level is by command. At this level the command name and its related arguments are read using one of the formats discussed above. After being read the command name is validated and to the extent possible parameters are syntactically checked. Finally the appropriate tool is invoked and supplied with the arguments.

At the second level of interactive command processing the command validation and parameter syntax checking take place seemingly as each field is entered. The second level is a large jump from the first, primarily because, the terminal handler must deal with input at the lexical rather than line level. That is, instead of buffering and sending symbols based on the end-of-record (carriage return/line feed), the handler must pick out the words or lexical elements of a command and send them one at a time. This level of interaction can provide almost immediate feedback regarding the syntactic validity of each command name or argument entered. I can clarify the difference, although not in total fairness, between the first two levels with the following example.

```
COPY S$ARCH.PAS TO BSEARCH.PAS
```

Assuming that $ is an illegal filename symbol, the first level of interaction would reply at completion of typing the entire line:

```
INVALID INPUT FILENAME: PLEASE REENTER COMMAND
```

However at the second level of interaction, the message might appear immediately after entering the invalid filename, and instead of requiring the entire command to be retyped might ask for just the input file.

```
INVALID INPUT FILENAME: PLEASE REENTER INPUT FILE AND REMAINING ARGUMENTS
```

The second level clearly has the advantage that in long commands the entire command need not be retyped. Unfortunately, the advantage is often outweighed by disruption of command entry.

The third level of interactive command processing jumps beyond syntax checking of arguments to combine logical checking. The implication at levels one and two is that some application specific software screens input to a level of syntactic validity and then ships the entire package of arguments to the appropriate command logic. At level three, however, the two functions of obtaining arguments and processing the command can take place concurrently. An example to clarify this is:

DELETE SEARCH.PAS, BSEARCH.PAS, TREESEARCH.PAS

The intention is that the file specified bv SEARCH.PAS may very well be deleted from the filesystem before the user has completed typing the file specifier BSEARCH.PAS. There are, of course, certain commands where this capability is not practical because of the data dependencies in processing the arguments.

Two additional levels of interactiveness should be specified for command processing. These are the logical equivalents of levels two and three in which the communication between the terminal handler takes place at the symbol level. Although communication at the symbol level is in some instances useful (visualize the KAPSE interface control functions), it is not in general a human-usable form of command processing.

To provide a Kernel Ada Programming Support Environment (KAPSE) that is useful in a varitey of situations, the KAPSE interface must be hierarchically structured. The hierarchy could be structured in such a way as to allow variable access by APSE tools. The form of the hierarchy would be based on different dimensions and each would have possibly many levels. The implementation of a given APSE would require a prespecified level for each of the kernel's dimensions. The dimensions could be based on such things as architecture, host/target configuration, and level of interactiveness of the APSE's debugging aids. When implemented in this way the transportability of the APSE tools and data bases would be limited only by Kernel levels. Additionally, APSE tools would be free to be fashioned to best utilize local conditions.

Program Construction. In conventional interactive systems the communication between editing, translation, and source level debugging functions is at the program level. One uses an editor to create a program, invokes the translator to obtain an executable representation of the program, and then invokes an executor or online debugger to test the program. A much higher level of interaction can be achieved when communication between these functions is carried out in units of source statements. When this is the case, no need for context switching from editing to translating to linking to debugging is necessary. All pertinent commands can appear to be handled by one tool, and translation can be completely hidden as a function carried out by the editor (program constructor). With this approach language statement entry or modification can be intermixed with controlled program execution. The user has the ability to execute parts of a program before the entire source has been entered, and the user has the ability to examine and modify either data or program parts as the program is running. A sample set of commands that would support such a system include:

enter_statement
insert_statement
change_statement
delete_statement
execute_statement
execute_next_operation_of_statement
execute_to_assert_break
examine_data
examine_statement

In many of the instructions given above it is necessary to specify a positon within some source statements where the instruction is to apply. In practice, systems possessing the capabilities suggested here commonly are crt-based, and the position is specified by a combination of paging and moving the cursor adjacent to the desired statement.

Program construction facilities possessing a degree of interaction equivalent with that discussed above and construction facilities with a much lower level of interaction must both be accommodated by KAPSE. The effect that this range of need has on the KAPSE interface is worthy of consideration. One representation of computation that is based on Johnston's Contour Model (see SIGPLAN February 1971) could be easily ammended to accommodate a high level of interaction. In such a model, relationships between source statements, documentation, and intermediate level statements must be formed. Further, special algorithms providing services that modify the entire computation when source changes are made as insertions and modifications are needed. Unfortunately, such a tight correspondence between source, internal form, and documentation is not necessary in a MAPSE.

Related Studies

A group of five faculty and twelve graduate students from the departments of Computer Science and Human Factors are involved in a long term human-computer interactions research project at Virginia Tech. Topical areas under investigation include an automated system for designing and managing human-computer dialogues, humanized software engineering, dialogue display principles, voice input and output, and human-computer task allocation. Studies in the areas of managing human-computer dialogues and studies in humanized software engineering seem to be highly related to the purposes of the KIT; two of these, involving automated design tools and levels of interactiveness, are now discussed.

Automated software design tools are gaining popularity primarily because they often force system designers and implementors to follow more closely a specific design methodology. Certainly this can do nothing other than improve the software production process, but the use of this type of tool has not been studied from two important points of view. Does the use of automated design tools hinder or enhance programmer/analyst performance? When online documentation and related design materials are available is continued support and use of the system eased? To evaluate these questions an interactive language designer, graphical editor, and related documentation aids are being constructed. The resulting tools will be used in experimental studies in which subjects will design, document, develop, and support software using the automated tools. The tools provide a means to create objects (design, documentation, or source) and establish

relationships between those objects. For example, a graphical editor may be used to create design documents that pictorially describe the workings of a system. The language designer may be used to develop a human-computer interface. And using the documentation aids, one may choose to form a relationship between a specific abstract data type and a section of the design document that describes how it is used or what it does.

Another set of studies being conducted in the humanized software engineering area of the project center on the learnability and use-efficiency of tools possessing different levels of interaction. For these studies we are using a Pascal based programming environment that can be configured to have different levels of interactiveness. We are providing subjects with one version of the programming environment, called PEEP, that supports highly interactive program construction in much the same way described previously. That is, the subject may enter a line of Pascal source and then if appropriate, cause that line to be executed. In general, the highest level of interaction provides the capabilities of assertion breaks with code and data modification mechanisms. Other subjects are presented a version of PEEP in which the level of interaction is equivalent to that on a batch system. Our hypothesis is that the higher the level of interaction the more efficiently experienced users compose software. That is, for someone who programs frequently enough not to forget a complex command structure, use of a highly interactive program composition tool optimizes production time. But, a tradeoff is involved since highly interactive tools are much more difficult to learn to use. Consequently, in situations in which programming is infrequently done, a less complex command structure and lower level of interaction is preferable.

Ackowledgments

This research was supported by the Office of Naval Research under contract number N00014-81-k-0143 and work unit number SRO-101. The effort was supported by the Engineering Psychology Programs, Office Of Naval Research under the technical direction of Dr. John J. O'Hare. Reproduction in whole or in part is permitted for any purpose of the United States Government.

MANAGING TRANSPORTABLE SOFTWARE

C. Douglass Locke
(IBM)

As the cost of developing software continues to escalate, and the availability of data processing personnel continues to decrease, it is becoming more and more clear that effectively transporting software packages from one environment to another is not only an answer, but is possibly the only answer to the problem of controlling these skyrocketing costs. Transporting software has been a subject of discussion in software engineering circles for years, but the recent economics of the situation have significantly increased the importance of this discussion.

From the outset of the Ada effort, one of the primary goals has been the capability to develop libraries of software packages which can be transported from one target system to another target system in which similar functions are required. The fulfillment of this goal, however, is much more difficult than simply generating a language in which transportable software can be written. It is the goal of this paper to describe some of the other aspects of transportability which are important to the development of very large software projects; software projects in excess of about 100,000 lines of source code.

What Do We Mean By Transportable Software?

At the outset we must define what transportability is, since it is used in many ways by many people. Transportability in this context will be used to denote the creation of software packages suitable for being executed on different target machines in environments other than that for which they were written, but in which the run-time environment is compatible. The meaning of compatible in this context will not be rigorously defined, but is definable for each project contemplating the use of transportable software. An example of such a package could be an aircraft navigation program for which the target computer is not the same as the one for which it was written, but the navigation equipment is the same (e.g. the

inertial navigation system, Doppler navigation system, attitude reference systems are compatible). It is conceivable that such a package could be defined even where some of those devices are not compatible but are generically equivalent (i.e. two different attitude reference systems or two different Doppler navigation systems). It is very clear that in the aerospace environment, many navigation programs have been developed for many different aircraft which are functionally identical; they compute aircraft position, velocity, and other parameters from very similar or even identical devices. This software has been developed over and over again, and must be maintained in every case separately at a staggering cost to the user community.

It is further assumed that the only use of transportability with respect to Ada programs is in the transportation of either packages or tasks. While it is certainly possible for lower level configurations of Ada code to be transportable, it is outside of the domain of this paper to consider this possibility because of Ada's ability to control their use and maintenance via the separation of interface data from the body of the code.

Achieving Transportable Software

Actually achieving transportability is, of course, a non-trivial proposition. The issue of transportability dates at least as far back as the definition of the FORTRAN language. While existing programming languages have not effectively supported the level of transportability that was a goal of their design, the FORTRAN language still provides significant experience in transportability. In FORTRAN it is possible to write code which can be transported from machine to machine, given the use of an appropriate standard subset (i.e. ANSI FORTRAN-66 or FORTRAN-77). In fact, we have an excellent example of the use of transportable FORTRAN code in the implementation of the CMS-2M support software by the Navy written in ANSI FORTRAN-66, and which has currently been successfully transported to at least six different host computers and operating systems.

Unfortunately, a common characteristic of the use of higher order languages to achieve transportability has been that, in general, inefficient code was produced. This relates to the definition of the higher order language itself and its non-suitability for certain problems. Again, the CMS-2M support software is a good example in that its run-time

efficiency is very low, because FORTRAN is not a particularly suitable language in which to implement compilers and other similar support software. The problem of creating a language which is transportable, and which has at least the potential of producing efficient code, has been the problem addressed in the Ada definition. Ada was designed to be efficiently compilable for many target architectures so that it would be usable in writing transportable code.

Not all Ada code, however, will be transportable. Technical issues of transportability include such things as standards of code generation in the use of transportable constructs: the use of direct machine code, the use of data types for which representation specifications can be written which can be implemented on various architectures, avoidance of the extremes of individual architectures, etc. It is assumed in the following discussion that this set of problems have been solved.

Management of Transportable Software

Given that these problems are solved, however, a number of management issues must not be overlooked. In developing a large software system, there are two basic phases in which transportability must be carefully managed:

A. Project proposal - the process of accurately estimating and bidding the software effort. The concept of transportability will materially affect the estimate of software development cost. This is, in fact, why we are interested in using transportable software in the first place. However, the proposal manager producing the bid must keep in mind that since he will not be producing the package or packages which will be transported from other systems, his ability to accurately measure the the quality and usability, resource utilization, as well as the interface and representation specifications used or assumed by those packages, will be crucial during the bid process.

B. Project performance - During the development of a large software system, tracking cost and schedule (e.g. earned value) is critical to the successful development effort. This is

a complex and difficult undertaking in any case, as evidenced by the abundant overruns and other evidence in this industry, but the use of transportable software significantly exacerbates the problem. During the software estimation process, some productivity rate (e.g. X lines of executable code per unit time) is assumed for code to be developed. Because of the expected low cost of the transported software, the effective productivity is very much greater for this portion of the program. This results in high leverage with respect to the development cost, producing not only large potential for savings, but the risk of a large overrun if this productivity level cannot be achieved. Thus it is essential that packages transported into a new system be monitored very closely to track the actual costs, so that corrective action can be undertaken as early as possible.

Even on the transition of existing large systems from one development phase to another for which it was expected that previously written code could be used in the new phase of the system, and in which the target machine and the language were unchanged, where the peripheral hardware is unchanged, and even the personnel are essentially unchanged, it has frequently been found that the level of transportability was much less during the implementation of the program than was estimated during the bidding process. There are many reasons for this, including poor initial estimates of the transportability, changes in the definition of the program after the bid is complete, unanticipated hardware changes which might have been small had the program been a new development but which were larger than expected because the program required significant changes, and the existence of software errors in the program to be transported, necessitating redesign or unexpected modification.

Issues such as these can strike fear into the most courageous program manager or proposal manager. Bidding is at best a difficult process, particularly on a very large program, and the unknowns produced by the desire or requirement to use software from another program are difficult to control. The current state of the art in measuring the quality of a program is not far advanced. Similarly, the state of the art in developing quality software has also been shown to be not far advanced. Therefore, the issue of quality measurement as well as the issues of readability and maintainability in the software to be transported become a key factor in cost estimating the use of transportable standard packages.

The solution to this problem is not yet clear. What is clear is that the problem, while not new, will be aggravated by the existence of large amounts of transportable software in a marketplace currently dominated by custom built software. The solution will probably take the form of guidelines, standards, and other controls to make quality measurement more accurate, accessable, and therefore, trustworthy. Transportable software will be written by two groups: those who set out to write transportable software, and who therefore will take precautions to observe such guidelines, and those who are not intentionally writing such software, and will not be as motivated to observe them. However, procurement agencies will be well advised to seriously consider the impositions of such guidelines in either case, since history is especially clear on this point: old software never dies, it just stops being maintained.

As we move into the use of standard packages of Ada software, it is the issue of quality control and measurement which will begin to dominate the transportability issues. Any guidelines, standards, or other constraints placed on transportable Ada software must be aimed at the goal of controlling and measuring software quality. Without this focus, Ada programming will be just another stone in the long path toward software transportability, producing a step forward perhaps, but falling short of the mark.

TRANSPORTABILITY OF TOOLS AND DATABASES BETWEEN APSES

Timothy G. L. Lyons
(Software Sciences Ltd, United Kingdom)

Introduction

The present situation of the design and development worldwide of a number of substantially different Apses means that Ada tools and data are not readily transportable between different installations, where these are using different Apses.

Nevertheless, there are clear requirements for transportability. Firstly, the existence of a standard language (Ada) provides strong incentives for the development of tools; this is decreased if they cannot be made available to the whole Ada community and could create a barrier to the development of large or sophisticated tools. Secondly, there will be life cycle requirements to move projects between different Apses, for example where a system is developed at one installation and maintained at another or where parts of a project are developed at different installations (where use of two different Apses may be inconvenient or undesirable).

Issues and Problems

The Kapse is specifically designed as the tool portability interface. When this interface is not standardised, there will clearly be severe difficulties. Some specific issues and problems are discussed here.

It must be emphasised that for each issue, it is the general case which is considered. In any particular case, a tool may use facilities in such a way that some specific portability mechanism can be devised, but this does not provide a general solution. In

addition, the interactions between these issues means that a simple solution to one problem may make another problem even more intractable.

Implicit Interfaces

In principle, the interface to which any Apse tool works is defined by the Kapse primitives, but in practice, these only define the means of access to the whole context in which a tool works. The ways in which tools depend on their entire context include:-

- implicit use of system facilities (e.g. large address space).
- implicit use of Kapse facilities (e.g. locking of an object during traversal of an associated database path).
- data structures in the database.
- other cooperating tools.
- hardware dependencies (e.g. VDU or printer control codes).
- data formats.

The first point is a general one which can apply even for transfer of programs between two Apses which have the same Kapse interface. Implicit use of Kapse facilities indicates that the whole semantics of Kapse operations must be considered and not just the superficial description. This point is returned to when the database is considered further. Hardware dependencies is again a general point for any type of system. The other points are among those considered further below.

Input-Output

Since the standard Ada INPUT_OUTPUT and TEXT_IO packages are included in the Ada language definition, all true Apses will provide implementations of these packages. It might therefore be expected that tools whose only apparent means of communication with the Kapse was through these packages would be readily transportable from one Apse to another. However, even in this case there may be problems:-

- if the tool uses STANDARD_INPUT and STANDARD_OUTPUT, these are predefined to be open when the tool starts to execute, but if it attempts to OPEN any files then implementation dependencies arise through the file NAME parameter.

- the overall form of files as sequences of elements is defined in the language, but there may be agreed conventions for the use of the elements (e.g. formatting conventions) which hinder portability.

Program Invocation

Differences in the facilities for program invocation may mean that some subprograms are not legal main programs on some Apses, as well as causing difficulties in parameter passing.

Use of Database

Examination of the current Apse designs shows that the database is the area where there is the greatest or most complex problem of incompatibility.

The differences between the whole philosophies of approaches to the database area cannot be emphasised too strongly. It is not just a matter of the syntax or form of an operation, but that the entire semantics are grossly different between different

Apses, since these semantics are based on very different underlying models. Thus operations which are superficially similar may have different semantics. Examples of the underlying differences are the extent to which structure is enforced by the database system itself (e.g. by reference to a schema) as distinct from structure imposed by tool action, and also in the whole approach to storing information and checking for database consistency.

Conceptually, what would be required to move a tool would be to find a mapping from the facilities of one database to those of the other. In practice, depending on the relative "power" of the two databases, this will require not only use of similar features in similar ways, but also simulation of missing features by explicit code in tools. Given the discussion above about the radial differences between database philosophies, it will be clear that in general this would be either very difficult or infeasible.

It must be reiterated here that it is the general case which is considered - in this case a tool which makes fundamental use of the database facilities. In simple cases, the problems may diminish.

Movement of Data

If the requirement is simply to move a tool or toolset from one Apse to another, then movement of data may not be required, but this will be a major task of a project and all its support facilities are to be moved. If the data is not strongly interrelated and makes little use of the full database facilities, then some mechanism might be devised, but in the general case there are many difficulties:-

- as discussed above a mapping between facilities is needed, and this may be infeasible.

- some of the structure may be held implicitly in the data by means of agreed inter-tool conventions, and these would need to be recognised by any tool performing a transfer.
- structure which is dependent on inter-tool or other conventions may be invalid or inconsistent.
- the database may have "educated" itself to accept partially inconsistant or invalid data (Pea 78).

Full Integration of Tools

Whilst there may be various ad-hoc ways to get a single foreign tool to run on an Apse, this may not integrate it into the Apse. It may not obey the Apse conventions, it may not interface correctly with other tools already on the Apse, or with other foreign tools from another Apse. The objective of an integrated environment is not achieved, and the full potential of the available Apse facilities is not used.

In short, if the environment is arranged to match the tool, the full benefit of the environment is lost.

The only way to achieve a genuine match of the tool to the environment is by re-writing the tool, and this may also entail some redesign to use the Kapse facilities. Such extensive re-writing is clearly not portability.

A Possible Approach

There are clearly major difficulties with any general concept of tool portability between different Apses, however there may be benefits which are achievable without the need for a general solution.

Tools may be divided roughly into two classes:-

- those which have extensive interaction with, or manipulation of the database (for example tools which interact with complex database structures, or configuration control tools).
- tools which have a very simple interface with the database, perhaps only accessing file type attributes using standard Ada input-output, and perhaps only having a single input and a single output.

In the first class, there may be a desire to encourage divergence (after all, this is, in a sense, why there is no common Kapse specification). The underlying designs may be said to differ because of differnig opinions on how best to organise structures in the database. Experimentation to determine the virtues of the different approaches is desirable, and moreover it is quite inappropriate to consider portability of this class of tool, since they are inherently poart of the differing approaches.

On the other hand, for a range of common tools (the second class) it may be possible to define a standard partial or minimal Kapse interface to which all such tools would be written and which would therefore allow portability of such tools. Such a partial interface might include access to file attributes of database objects using standard Ada input output, and perhaps also some limited form of program invocation.

It is important to emphasise that such an interface must be defined early so that tools can be designed specifically to match the interface, rather than matching an interface the tool already uses to the standard. This is to ensure that the tool is fully integrated into the (partial) environment. Also certain Mapse tools (e.g. the editor) must be designed

to match the established conventions so that they will co-exist happily with the portable tools.

Such an approach would of course be standardising on a subset Kapse. It could be argued as to whether the benefits of standardisation at this level outweigh the disadvantage that the subset of tools are not making full use of Kapse facilities.

Regardless of the approach that is adopted it is hoped that this transportability effort will take a wider view of Kapse features than just the ALS and AIE systems, so that its applicability could include:-

- the Apse developments in Europe.
- future evolution and enhancement of the ALS and AIE.
- any other Apse systems which might be developed in the future.

References

(Pea 78) Pearson D.J., A study in the pragmatics of operating systems development. International symposium on operating system techniques. Paris 1978.

(UK Ada 81) United Kingdom Ada Study. Final Technical Report. Vols.1-7 Available from The Librarian, Division of Numerical Analysis and Computer Science, National Physical Laboratory, Teddington, Middlesex, England.

PORTABILITY AND EXTENSIBILITY ISSUES

Charles S. Mooney
(Grumman Aerospace)

Introduction

The Stoneman document placed no small amount of emphasis on portability- mentioning it explicitly on the first page and then proceeding to specify a means of providing such portability - the isolation of machine dependencies into a well defined kernel. Portability is an extremely important goal for any APSE or APSE component and will, to a large extent, determine user acceptability. Portability is important to the industry in general, and a major test of the ALS and AIE will come when the government releases them and users start to adapt them to their own particular hosts and corporate or gov ment environments. The design decisions made by the contractors for these two efforts will be reviewed and revisited by the industry at large for a number of years to come.

My own organization would like to see a truly portable software environment. We currently are responsible for five major military aircraft- four Navy, one Air Force. Amongst these aircraft we have 16 different on board computers - mainly programmed in assembly language and supported by spectrum of hosts ranging from the IBM 1800 and the Litton L304 through 370's and CYBERs. Portability, and more generally, reusability is of prime interest to us.

Portability Among Different Hosts

There are several viewpoints from which one can address portability. The most apparent is that of MAPSE portability in moving from one host to another. This aspect is the one most directly addressed by the MAPSE/KAPSE scheme in Stoneman. It appears to pertain most immediately to the rehosting of the MAPSEs that are currently being built and, by extension, to host-executing tools being added to those environments. Another viewpoint of APSE portability is that of applying an existing APSE to a new target. In addition to the need for obvious items like a new code generator, package STANDARD, and a run time package, potentially, special linkers and debuggers may also be required.

One major area of portability that is not always obvious is that of people portability. One of the benefits of a standardized programming language is that it should eliminate the need for retraining programmers when they move to a new computer. A standard MAPSE should also provide this type of benefit, at least on the surface of the user interface. Without control, there may very well develop a myriad of APSE command languages. An extreme example could be that of having different command languages for the same host when it runs under different operating systems. There does not appear to be any justification for requiring a user to maintain a working knowledge of multiple command languages, all of which purport to invoke identical functions. One recommendation which has been offered before by others and which is worth repeating is that of a Stoneman Appendix which dictates a standardized user interface. Perhaps if this recommendation were to be adopted, the ACVS could be extended to include some level of testing of the command language.

Portability Among Different Environments

A broader view may be taken of portability - that of moving collections of software tools from one software environment to another. These environments would differ in their purposes, in the type of applications they support and in corporate of government agency specific requirements. Adapting MAPSE within these environments will require embedding of existing tools within them. Examples of such tools could be agency specific documentation or configuration accounting packages and programs to provide interfaces with corporation specific progress reporting and tracking systems. Large corporations have investments in these types of tools and, to some extent, methodologies built around them. Government agencies have frequently adopted the use of such tools to maintain weapon systems. Introducing Ada will mean installing a body of existing software into an Ada environment or, alternatively, the development of interface software to extract from MAPSE data base the information needed by existing tools.

Perhaps additional items have to be standardized. It would be desirable to have APSE extensions, such as some of the tools mentioned above, work regardless of the MAPSE they are added to. It does not appear that this will be that easily accomplished with the two MAPSE currently in development. Separate MAPSE specific interfaces would have to be generated for each tool and this may lead to an untenable counter-portable situation at some time in the future. What might be a solution would be the standardizing across MAPSE of a pertinent data interfaces to permit easy enhancing.

Another area of concern related to the general issue of portability and extensibility is that of the user interface. Assuming that MAPSE commands were controlled by a published specification and an ACVS extension, there could still exist a potential Tower of Babel in all the extensions

that different organizations would add the their environments. To lend some possible order to what might be a chaotic situation, perhaps this group or the KIT could establish some guidelines to tool developers for extending the command language. One thing that definitely should be done as a task by the MAPSE developers, is to provide detailed instructions on how to add tools to their MAPSE - how to add commands, how to place the tools in the system, etc. It is a requirement that adding tools should be easily accomplished and also that the normal anticipated MAPSE system maintenance should be capable of being done with little interference to (or buy) any added tools.

Concluding Remarks

Normally a paper ends with conclusions. Given that this is our initial meeting, perhaps it is more appropriate to end with a question. The Stoneman document acknowledged that it was incomplete in certain areas. Reference was made to appendices that could be added to specify certain interfaces in more detail. Are there any efforts underway, or plans to start such efforts, to build these appendices? Will the ALS and AIE contractors be tasked to adhere to these appendices?

Glossary of Acronyms

ACVS - Ada Compiler Validation System

AIE - Ada Integrated Environment

ALS - Ada Language System

APSE - Ada Programming Support Environment

KAPSE- Kernel Ada Programming Support Environment

MAPSE- Minimal Ada Programming Support Environment

PORTABILITY AND KAPSE INTERFACE STANDARDIZATION ISSUES

Ann Ready
(Planning Research Corporation)

Introduction

The official standardization of the APSE and KAPSE is being delayed because of an industry wide lack of experience in the area of building programming support environments. That is, programming support environments are a state of the art phenomonon: much discussed, but not well understood. However, if the Navy is to succeed in integrating tools from APSEs of differing designs and in transporting data between APSEs of differing designs then some minimal standards for KAPSE interfaces must be established. The issues to be considered in arriving at these standards include both upfront, design independent issues, such as an industry wide consensus on the functions that should be supplied by an operating system and an Ada community wide consensus on the extent of the data management services to be supplied by the KAPSE, and detailed design issues, such as how to standardize interfaces in the KAPSE and MAPSE in order to insure data portability. The purpose of the paper is to raise and discuss some of these issues.

Design Independent Issues

There are several design independent issues that must be settled before other KAPSE interface issues can be discussed. These design independent issues involve a refinement or clarification of the KAPSE level functions described by the Stoneman specifications. Two important design independent issues involve reaching an industry wide or at least an Ada community wide consensus on the primitive functions that should be supplied by an operating system and by a DBMS. A consensus in the area of operating system functions is necessary because the KAPSE functions as an operating system from the point of view of the MAPSE and APSE tools. A consensus in the area of DBMS functions is necessary since the Stoneman specifications imply (5.A.9) the existence of some type of data management functions in the KAPSE even though the phrase DBMS is not used. In fact, the Ada community must reach a consensus on exactly what level of data management functions is necessary to the APSE as a whole and which of these functions should be provided by the KAPSE. Without a consensus in these areas, it would seem that any KAPSE interface standardization efforts would be premature and possibly doomed to failure. The best that could be expected with no consensus would be that standards would emerge within each major school of thought and that tools would be transportable among APSEs designed by the same school only.

In the area of operating systems, there appears to be some industry consensus emerging. The growing popularity of such "portable operating systems" as UNIX*, which is now available in some form on systems ranging from mainframes to micros, points in this direction. In the home computer market, several of these portable operating systems (e.g. CP/M*) have become accepted standards with incredible rapidity. Thus the idea that there should be a standard set of operating system functions is becoming widely accepted. Since the KAPSE interfaces supply a logical operating system interface to the APSE and MAPSE programs, a KAPSE interface standardization process can take advantage of both the emerging industry consensus and the growing body of experience in porting operating systems from one machine to another.

In the area of DBMSs, the industry seems, at present, to be in a state of flux. Relational DBMSs have received much recent attention and have in their favor a potential flexibility and application portability lacking in their network and hierarchical counterparts. However, operational experience with fully relational systems is still reasonably small, and some of these systems seem to have response time difficulties, especially in the area of "production" work as opposed to ad hoc queries. Many of the network and hierarchical systems are adding relational style interfaces to increase their flexibility for ad hoc queries while retaining their response time advantage for production work. Thus it is possible that the DBMS of the future may be a hybrid of both the relational and the network and hierarchical systems, but it is still too early to tell.

The Ada community must agree on several issues which involve DBMSs, including the type of interface (relational, network, or other) to be supported. The Ada community must agree on the extent of the data management services required by an APSE and on the APSE level where these services must be supported. Currently, there is not much consensus on where the data management services are to be supported. Some APSE descriptions show the DBMS as a MAPSE tool; others show it embedded within the KAPSE. There does not seem to have been much discussion on whether the DBMS needs to be a fully general one (the Stoneman specifications are vague) or on the requirements which the expected size, complexity, and style of use of the KAPSE will put on the DBMS. It is to be hoped that the special requirements of an APSE will place enough constraints on general DBMS issues so that the Ada community can reach a consensus. The situation is complicated by the fact that portability issues are also involved. These portability issues are further discussed with the design dependent issues.

There is a third design independent issue which must be carefully dealt with if KAPSE interface standardization is to be successful. This issue is in the area of metrics or judgement criteria. Before the KAPSE interfaces can be standardized, an explicit set of criteria must be developed for judging various interface designs. Such criteria must include both portability measures to judge the "average" difficulty of implementing KAPSEs to support the given interfaces, and use measures, such as maximum acceptable response times for critical functions and ease of implemen-

---

*UNIX is a trademark of Bell Laboratories
*CP/M is a trademark of Digital Research

tation of critical MAPSE tools. These criteria may be difficult to develop, but it will be possible to draw on the growing industry experience in porting operating systems and on the small but growing industry experience in developing programming environments.

Design Dependent Issues

Some of the important issues which involve specific aspects of KAPSE interface design concern questions of data base portability. The KAPSE interfaces should be designed in such a way that not only are the APSE and MAPSE tools readily portable, but that the KAPSE data base information is also readily portable. That is, if a system of programs is developed on one APSE, then the information collected about that system of programs should be transportable to the machine on which those programs will run operationally, even if that machine has an APSE of a different design. Tool and data base portability issues involve a variety of interrelated problems.

One of the problems in dealing with a DBMS is that it has two interfaces: one to the application programs and one to the host file management system or operating system. Since an APSE DBMS sees all the APSE and MAPSE tools that access the data base as application programs, the application/DBMS interface would appear to be a KAPSE interface. Standardization here would help insure the portability of these "application programs" to other environments with the same interface. However, when considering the portability of the data itself, at least as far as network or hierarchical DBMS is concerned, it may be preferable to standardize the DBMS/OS interface as a KAPSE interface to insure the portability of the DBMS along with the data. Data is difficult to transport between differing network or hierarchical systems because of the tedious hand translation of the data base schema which is usually necessary. This translation is used to make adjustments for the various ways that different network and hierarchical systems represent different relationships. These portability difficulties are proportional to the complexity of the data base. Relational systems seem to suffer less from these difficulties. Even so, the importation of new data containing new relationships will necessitate the renormalization of a relational data base.

It is not clear how to reconcile the conflicting reasons for standardizing the applicational/DBMS interface and the DBMS/OS interface. If the DBMS/OS interface becomes a standardized KAPSE interface, then "hybrid" APSEs will be in the awkward position of supporting two (or more) distinct DBMSs and data bases. This duplication of function could be a strain on system resources, and certain tools, such as configuration managers and management report writers, would be very difficult to write in such a system if they had to access information in all the data bases. Standardizing both the application/DBMS and the DBMS/OS interfaces might have the undesirable side effect of unduly restricting the flexibility of the DBMS or prematurely standardizing the DBMS. Any compromise position, such as making the KAPSE level interface a high level file manager interface, probably has all the same disadvantages as standardizing the DBMS/OS interface.

Even requiring a relational DBMS does not solve all the problems associated with tool and data base portability. To ensure the portability of data base information, a standard intermediate form will be necessary. That is, all APSEs which expect to import or export data base information should have programs which can load data into the data base from a tape file in the standard form or unload data from the data base onto a tape file in the standard form. This standard intermediate form and these load and unload programs must be designed with care to insure that no information is lost in the transportation process. The Stoneman specifications (5.B.1.(b)) imply that attributes can be implicitly associated with a data base object throught the inclusion of the object in a directory. Other problems can arise if the transported data contains relationships not expressed in the existing data base or vice versa. In this situation, even a relation data base must be renormalized to prevent information loss. Even so, it may not be possible or desireable to construct the new relationships in the part of the data base which did not have them before. Yet, if these new relationships are not established, APSE and MAPSE programs may no longer work correctly if they expect such relationships to be established for all objects of given categories in the data base.

Thus, there is a problem of information integration as well as of information loss. This problem is made more acute by the fact that the Stoneman specifications allow absolute freedom in the area of the attributes which may be associated with data base objects and the resulting associations which can be made between objects. If data base information is to be fully transportable between APSEs, then it is possible that some restrictions will have to be made on the types of attributes and associations that can be made. The basic problem involved is that it is difficult to map a complex set of associations into a set of simple ones or into another complex set with incompatable features. The difficulty in placing restrictions on associations is in being certain that the allowed set of simple associations is rich enough to build all the "desirable" complex ones.

Summary/Conclusions

This paper has attempted to raise some of the issues and problems surrounding the KAPSE interface standardization process. A special attempt has been made to focus on a few of the issues involving DBMSs and data base portability. The issues raised can be viewed as falling into three categories: metrics, refinement of Stoneman specifications, and restrictions on Stoneman specified freedoms. In the area of metrics, a whole set of criteria need to be developed for judging KAPSE interface specifications. Little work has been done in this area. In the area of refining the Stoneman specifications, a consensus must be reached on the exact set of operating system functions to be supported by the KAPSE and on the type and extent of the data management services needed by an APSE and at what APSE level these services are to be supported. It appears quite likely that an agreement can be reached on the set of operating system functions, but there is currently little consensus in the Ada community and in the

industry as a whole on many of the data base issues. The choice of which DBMS interfaces to standardize and which to select as a KAPSE interface involves a variety of tradeoffs between tool and data base portability and data base integration. The final area of restrictions on Stoneman is also centered on issues of data base portability and data integration. Tools not only depend on standard content of data base objects (e.g. that the compiler front end output is in the form specified by Diana) but on standard relationships between data base objects (e.g. the connections between the library units which form a configuration). Some of these relationships are specified in Stoneman, but Stoneman also gives complete freedom to form other relationships. Some restrictions will have to be placed on this freedom so that the relationships expected by one set of tools can be readily mapped onto the set of relationships expected by another set. Since the data base is the glue which holds the APSE together, these issues are of great importance.

MAKING TOOLS TRANSPORTABLE

Sabina H. Saib
(General Research Corporation)

Abstract

This paper describes some approaches to making software tools transportable, whether across machines or across operating system. A multipronged approach is described which includes restricting the contructs used in the tools to those found in the early versions of Ada compilers, limiting machine-dependent subprogams and data to a single package, and defining simple virtual data structures and a set of low-level functions to operate on the data structures.

Introduction

Tools have been developed which have been transported across a variety of machines and operating systems. Table 1 lists a number of these tools which have been developed by the General Research Corporation. The techniques used in making the tools transportable can be used in the development of Ada tools to make them transportable across the various Ada compilers and Ada environments that will be available in the near future. The approach taken to make tools transportable is one of enforcing simplicity in the design, in the implementation, and in the use of input and output.

The methods presented are pragmatic rather that idealistic. That is the methods assume that Ada compilers will be in development for some time to come with early compilers and environments being delivered incomplete and with known problems.

In determing the transportability of tools, we should not neglect the medium of transporting tools, data, and programs. In the past, to guarantee some degree of transporting tools, we have written tapes which

TABLE 1

GRC SOFTWARE QUALITY TOOLS

RXVP, FAVS, JAVS, CAVS, J73AVS

O INSTALLED ON VARIOUS COMPUTERS

H6180 IBM 360/370 CDC 6400/660/7600 CDC CYBER 173 UNIVAC 1108/1110 DEC PDP 11/70 11/780 ITEL AS/6

O INSTALLED ON VARIOUS OPERATING SYSTEMS

GECOS/MULTICS

MVS/TSO/CMS/VM

SCOPE/GOLETA/NOS/NOSBE

EXEC 8

RSX/VMS

O FOR MULTIPLE LANGUAGES

JOVIAL J3, J3B, J3B-2, J73 FORTRAN IV, FORTRAN V, IFTRAN, DMATRAN PASCAL, VPASCAL COBOL 74

are blocked in 80 character ASCII character images at low densities. While this method results in tapes which can be read by all systems after some effort is spent in resolving the parity bit usage of the system, it has the undesirable feature of wasting appoximately half of the tape. It would certainly be a worthwhile feature of Ada environments if they were able to read tapes which were blocked in some multiple of 80 character records such as 800 or even 1600 characters. It would also be desirable if the parity bit usage was defined. Even better would be a method in an environment to allow what is often called the "transparent" method of communication where one host could talk to another host or target computer to transfer tools, data, and/or programs over a hard-wire communication without requiring tape.

Construct Restrictions

One of the major detriments to tool transportability is the use of non-standard features or extensions to the language. Ideally, the use of Ada, which does not allow supersets of the language, will prevent this hinderance to portability. However, in any new language, it must be recognized that for some time to come a few of the features will not be implemented or will be implemented incorrectly. A tool builder can not be a compiler validator nor can a tool builder enforce compliance to compiler standards.

One possible solution to the fact that Ada will remain "incomplete" for some time to come despite the promises of the many competeing implementations is to restrict tools to use constructs which are available in the early Ada compilers. There are several such compilers to date which can be used as guides to what can be expected in early DoD sponsored compilers. Examples of such compilers are the NYU, TeleSoft, Intel, Western Digital, and Irvine compilers.

There are two ways to determine what is available in the early compilers. One way is to believe manufacturers' literature on the compiler. Table 2 is an attempt to merge two such lists of features available in February of 1982. The other way is run the existing

TABLE 2

UNIMPLEMENTED FEATURES IN TWO CURRENT COMPILERS

	MANUFACTURER 1	MANUFACTURER 2
DATA TYPES		
	OVERLOADED ENUMERATIONS	
		CHARACTER LITERALS IN ENUMERATIONS
		NUMBER DECLARATIONS
	DIGITS	FIXED POINT TYPES
	DELTA	
	DISCRIMINANTS	DYNAMIC ARRAYS IN DISCRIMINATED RECORDS
		ACCESS CONSTRAINTS
		ACCESS TO UNCONSTRAINED ARRAY TYPES
	DERIVED TYPES	DERIVED TYPES
	MULTIDEMENSIONED ARRAYS	
	PRIVATE TYPES	
EXPRESSIONS		
	SELECTED TASK COMPONENTS	
	ATTRIBUTES OTHER THAN FIRST, LAST, RANGE, LENGTH, POS, VAL, PRED, SUCC, IMAGE, BASE, ADDRESS, OFFSET	ATTRIBUTES OTHER THAN FIRST, LAST, RANGE, LENGTH, POS, VAL, PRED, SUCC
	ARRAY AND UNQUALIFIED AGGREGATES	DYNAMIC AGGREGATES
	OTHER THAN SIMPLE CASE OF ALLOCATORS	ALLOCATORS WITH CONSTRAINTS
	DERIVED TYPE CONVERSION	OTHER THAN NUMERIC TYPE CONVERSION
		LITERAL EXPRESSIONS

TABLE 2 (Cont.)

STATEMENTS		
		ABORT STATEMENT DELAY STATEMENT CONDITIONAL AND TIMED ENTRIES
SUBPROGRAMS		
	OVERLOADED ARGUMENTS	OVERLOADED OPERATORS CONSTRAINT CHECKS FOR OUT PARAMETERS
PACKAGES		
		GENERIC PACKAGES DEFERRED CONSTANTS
	EXCEPTIONS PRIVATE TYPES LIMITED PRIVATE INITIALIZATION	
VISIBILITY		
	RENAMING	RENAMING
TASKING		
	TASKING	TASK TYPES TASK PRIORITIES

TABLE 2 (Cont.)

SEPARATE COMPILATION	
AUTOMATIC ORDERING	
SUBUNITS	SUBUNITS
	SUBPROGRAM LIBRARY UNITS
EXCEPTIONS	
BLOCK EXCEPTION HANDLERS	
SUPRESSING OTHER THAN RANGE	
GENERICS	
LOCAL GENERICS	GENERICS
GENERIC WITH	
IMPEMENTATION DEPENDENCIES	
REPRESENTATION SPECIFICATIONS	REPRESENTATION SPECIFICATIONS
	PACKAGE SYSTEM
	MACHINE CODE INSERTIONS
	UNCHECKED STORAGE DEALLOCATION
INPUT OUTPUT	
PARTIAL	ENUMERATION TYPE I/O
PREDEFINED	
PARTIAL	REPRESENTATION PROGRAMS
	SYSTEM PROGRAMS

validation suite for the features against each of the present day Ada compilers. The second method is suggested as being more objective. In either case, an "and" of the features that are available can be constructed. To make portable tools, the constructs which fit in the "and" of the currently available features should be the only ones that are used. This means that portable tools should not use generics, aggregates, overloading and tasking since even chosing just two current compilers eliminates these features. To promote the development of tools which will be portable between compilers, a publication which lists which constructs are acceptable to all early compilers would be useful to tool builders. The publication could be updated on a yearly basis.

Table 2 does not identify the two manufacturers. The data was obtained from advertising literature and was not validated. It was noticed that although one manufacturer listed that all statement types were recognized, it listed that tasking was not performed. Inconsistencies of this nature are sure to arise when advertisements are used instead of the more objective validation suite tests.

Machine Dependence

Tools will need to make some use of machine dependent features in Ada. For purposes of portability these features should be restricted to a single package so that when the tool is transported the changes which are necessary to support the tool on a particular machine can be located easily.

Ada has certain statements which are for the use of machine dependent features. Examples are the FOR ... USE statement, the unchecked conversion feature, and the FOR ... AT statement. Certain constants in Package Standard are also machine dependent such as those giving the number of bits in a word and the number of characters in a word. Predefined data types such as INTEGER, REAL, and FLOAT are machine dependent in that the size of the numbers that can be represented is dependent on the implementation.

To contain machine dependence to a single package, it is possible to use a code auditor to detect the presence of machine dependent constructs. The predefined data types can be defined in this package to be of a size that is available on most 16 bit machines or on most 32 bit machines with a statement to the effect that the tool is efficient only on machines of a certain size. However the tool will still be transportable to smaller machines.

Design Restrictions

The object oriented design methodology provides help in making tools transportable. The reason it helps transportability is that it builds upon simple data structures and low level functions which are used throughout an application or tool. This is in contrast to a design methodology which allows a designer to implement numerous data structures and functions through out a tool without consideration as to how the structures could be made more general to satisfy needs in more than one part of the tool.

In applying an object oriented design methodology to Ada, we first need to define the Ada data types that are applicable to the tool. For example, what is the best length string. Then we need to define the data structures that can be useful throughout the tool. For example, what is the best structure for a tree or symbol table that could be used in several parts of the tool. Next we need to define the set of functions that can be applied to these data structures and elements of the defined data types. The tool will then build upon these low level data structures and functions to create higher level functions.

The arguments for object oriented design methodology are that it will save on maintainence cost and that it will result in a simpler design. One can view the transportability problem as a maintainence problem since changes often must be made when a tool is moved across systems. If the changes can be isolated to a few low level funtions, then the transportability problem is simplified.

Input Output Restrictions

Even the most transportable languages to date suffer from differences in the use of input and output functions. The differences can even come about when the operating system is changed without any changes to the language.

To minimize the problems due to input and output function incompatibities, it is recommended that input and output be treated as a machine dependent feature. That is, the use of Ada input and output functions should be restricted to a single package. The tool itself should cali upon these functions instead of the Ada input and output functions. This extra level of protection from the language provided functions will let the tool designer correct for the differences in different operating systems or environments in a single location. For example, one of the Ada systems may present a carriage return followed by a line feed at the end of a line while another sytem may present two line feeds at the end of a line and a third sytem may present a single carriage return. If throughout the tool there are tests for the first of these end of lines, the tool will be difficult to transport.

Where possible a tool should use human readable input and output even for intermediate files. This has the advantage that when the tool is moved, the installer will be able to read the input and output.

Summary

In order to promote transporting tools between computer systems, it is necessary to observe restrictions. Most of the restrictions will result in simpler programs and follow the guidelines for an object oriented methodology. Some of the restrictions recognize that although Ada will eventually solve some of the major transportability problems, for some time to come one should plan to avoid problems that have arisen in the use of other high level languages. By isolating potential problem areas to single packages, the transporter of a tool will not be forced to search through out the tool for problems.

INCLUSION OF DICTIONARY/CATALOG AND CONTROL FEATURES WITHIN THE ADA ENVIRONMENT

Edgar H. Sibley
(Alpha Omega Group, Incorporated)

Abstract

Two major areas impact on the transferability and reusability of ADA packages, tasks, etc.: the database management capabilities (including the DBMS, dictionary, and configuration management tools) and the information storage and retrieval (ISR) capabilities that make it possible to classify and retrieve information on useful packages--so that other ADA programmers and system designers can find and reuse them. The database management capabilities are somewhat primitive in the "Air Force" KAPSE specifications and nonexistent in the "Army" KAPSE; there is apparently no current attempt to provide any information retrieval and classification methods to the ADA environment. This paper deals with the need, architectural implications, and effect of these additional features on the KAPSE as it evolves in the next three years. It is our contention that such additional features are crucial to portability and reusability in the ADA environment.

Introduction

The design and implementation of modern information systems has benefitted substantially from the introduction of the database management system (DBMS), but there are certain inadequacies that have developed in their use:

(i) The proliferation of user names, making the system documentation confusing and requiring some function to aid in the control of names and adoption of naming conventions. The device that emerged is termed a "Dictionary System."

(ii) The control of the use of components that are undergoing change; the assembly of incorrect versions of modules or data into a system has caused gross problems in the past--sometimes when attempting to cut back to a previous "working" version of a system, sometimes in using the old file descriptions or job streams with the new files, etc.

One method of increasing productivity in the development of modern systems is the use of large-scale software packages (such as the DBMS) as a major piece of reusable software. Unfortunately, however, it is often difficult to find these possible pieces of reusable software or packages. The reason for this loss of power might be considered a class-

ification problem; i.e., if there is no well organized catalog with a well designed classification scheme as a basis, the material is "not known to exist."

In the light of these comments, there are four important interrelated tools that we feel are missing from STONEMAN and subsequent implementations:

(a) A generalized database management system (DBMS) specification that allowed effective access to data;

(b) the addition of dictionary facilities for control of data, program and the users/tools of the system;

(c) the addition of software configuration management control within this dictionary environment. This should be possible within "extensibility;"

(d) provision for a cataloging facility to enable software reusability.

General Control Tools

In our review of the general specifications of the ADA environment within the STONEMAN and Army documentation, we addressed the following major questions:

1. What method is used in the current specifications for the control of data files?

2. In what ways do these aid the development of future database management systems, dictionary systems, configuration management systems, and the use of information retrieval techniques for finding potentially reusable software and data?

3. Do current specifications make some of these operations difficult to add?

4. Should these additions be in the KAPSE or MAPSE?

In general, these specifications use older architecture concepts for library and other storage functions. Thus there appear to be problems of access control and diffusion of data control functions that will hinder the development of good data and storage facilities in the ADA environment. Undoubtedly, because some data operations are spread through the specification, there may be some difficulty in integrating them into a proper set of data modules which must be added, but this should be a relatively inexpensive retrofit if started soon (or left to the next phase with some cost). However, the crucial question appears to be one of location of the tool. If it is to be in the MAPSE, there is little problem, but if it goes in the KAPSE (as we shall propose is necessary), then there must be some redesign, refit, or deficiency in current implementations. Moreover, this suggests that the Navy implementation should start with a redefinition of its KAPSE architecture.

The major question that must be addressed in dealing with the KAPSE/MAPSE interface and the location of some special tools seems to be one of control. And here the term control is intended to include general concepts of security--though this paper is not intended to deal with that issue in any detail, but more to use some basic principle common to control of access and use, and techniques to aid in the application of general management principles in the production of software. Basically, the arguement is presented as the "weakest link in the chain;" viz:

> If the control of access through and with various major tools is not resident in the KAPSE, then a good programmer can circumvent the controls by writing for access at the KAPSE interface rather than through the MAPSE resident tool.

The question of control must, therefore, be addressed at all the various tools which might be expected to require limitation of use of software or data. A partial (i.e., not necessarily complete) list of these is:

(i) Security protection through password protection, etc.

(ii) Protection of hidden packages or data from disclosure to programmers, etc.

(iii) Limitation to access of data in a data sensitive rather than procedure sensitive mode (i.e., a person allowed to access any data via a given package may be deemed "procedure sensitive," but data that may be accessed through procedures only if the user and procedures have certain matching characteristics are deemed "data sensitive"--e.g., a person allowed to access any data within their squadron, but not outside it).

(iv) Provision of a dictionary that can allow programmers to communicate the data names, procedure names, and usage of both within the system packages that they are developing. Naturally, in order to retain the "hidden" nature of some parts, the dictionary itself must have internal controls.

(v) Provision of methods for enforcing good configuration management principles (in versions, testing, etc.) through the use of extensible features in the dictionary (as discussed briefly later). Again, the controls imply that the configuration management system be inside the KAPSE--or it can be circumvented.

(vi) Need to find data and packages using modern information storage and retrieval (ISR) techniques, probably involving good classification techniques. However, again this implies a need to control access, if hidden or secure features are to be protected from non-authorized access.

The KAPSE and MAPSE in Host and Target Architectures

It is probably obvious to many of the people in the ADA community that there must be differences between the Host and Target KAPSE. However, the problem is not apparently always understood by workers in the area; e.g., in the documentation for contract F 30602-80-C-0292: The Ada Integrated Environment; Interim TR 15 March 1981, the KAPSE Database System, page 3-1 states:

> "The KDBS strictly belongs to the MAPSE; it is designed for software development and has no place on a target machine."

The implication of this statement is possibly true in one respect--that a target machine may not need the full DBMS (KDBS) capability, but the other implication is that the KAPSE for a host machine is therefore not to contain the DBMS (KDBS). We suggest that this is not true.

There are, of course, other cases where the KAPSE will probably be different in host and target--one is in the debugging capability. Presumably, the target must provide debugging data during test phase--but during the operational phase, the debugging facility is probably not needed (depending, of course, on the mission). This suggests an architecture in which there are <u>levels</u> of KAPSE. We propose a need for at least the following:

(i) The KAPSE within the host. This is the "complete KAPSE."

(ii) The KAPSE within the target during debugging. This is close to the "complete KAPSE", but may not need some features such as a DBMS or Dictionary, and <u>especially the security associated with them</u>, because the testing is carried out under a controlled (secure) environment. It may be valuable to consider the "test machine" to be larger than the "operational" machine (see below), because further parts of the system can be excluded after test.

(iii) The KAPSE within the target during "operational" conditions. Those parts of the KAPSE that are needed for final testing can be excluded, with the machine able to have either increased workspace, or less memory, or other configuration in order to improve the payload or operational speed, etc.

Obviously, these are somewhat radical suggestions, but might well be proposed for future consideration.

The Use of Database Management Systems and Extensibility

(a) STONEMAN and the Army System

In looking at the current specifications for STONEMAN and the Army implementation, some comments arise as follows:

i. The "Environment Database" facilities deal with some issues that are really either DBMS or dictionary system (DS) related. If the DS were implemented using a DBMS, then the facilities would be consistent for both the data and the metadata.

ii. The consistent use of a query facility within the DBMS would imply that the same query facility could also be used in the DS. Thus all controls and language interfaces could apply to the DS.

iii. More effective access controls based on the DBMS and its control facilities through the DS(e.g., the ability to limit access based on data value and/or time, etc.) could augment the rather simplistic access controls in the current specification.

References in STONEMAN (and ALS and subsequent specifications that deal with the data facilities) are given on pages:16 (standard routine libraries), 22 (database storing objects with "version groups"), 24 (software configuration and history preservation in KAPSE), 24 (progress and statistical reporting), 33 (access, protection, and archiving), 36 (access control and other tools), 38 (compiler symbol table), 44 (configuration management in MAPSE), 46-47 (the library), 48-49(documentation, project control, configuration control, life cycle management, etc., in the APSE).

The above references deal with those interfaces to the Environment Database and the software that must be provided (as discussed in Appendix 50 of the A spec). The material is somewhat vague, and the question of relative need for software in the KAPSE, MAPSE, and APSE is difficult to infer: as an example, the use of backup facilities for reliability of the host and target are essentially ignored.

The question of the different roles of the KAPSE and MAPSE is not discussed relative to the roles of the host and target machine. It is reasonably obvious that the target does not always need the same protection as the host. In fact, if the usage of the target precludes any maintenance that does not initiate in the host, then all the configuration management should be retained only in the host, but this implies controls that cross over from host to target--thereby requiring some concepts of distributed data and processing systems (which also seem to be ignored in STONEMAN).

When the STONEMAN requirements are articulated in the A and B specs, the problem of lack of focus of the data and program control produces further problems. One example is in the method of "locking" the database files--they are to be protected (Appendix 50 of the A spec) by providing user identification or tool pseudo-identifiers) at the node level for the type of operation(read, append, execute, etc.). This is a solution, not a requirement and it ignores many problems of data-dependent access mentioned earlier. Indeed, it is obvious that the "hidden" data and processes needed the addition of the "tool identifier" to allow access by a user program to hidden material.

Possibly, this confusion of requirements with solutions to needs is the reason that the questions of renaming and deletion of nodes is so strange (Appendix 50 of the A spec and section 3.2.3.2 of the KAPSE B5 Spec). The question of how to determine whether a dependent node is shared, how to reference other nodes, etc. is discussed, but the method should be irrelevant--our solution would be in the dictionary/directory ensuring control through the dictionary system software.

Thus in this latter example, the specifications assume no central focus of data control and have to build in the functions at other points (sometimes specifying the method in too great detail, so that retrofit will be difficult).

(b) The Air Force System Specifications

Unfortunately, the three specifications of the three Air Force contractors also appear to be somewhat deficient. Detailed criticism of these works will not be included here at this time as they are in a somewhat "formative" stage and though the INTERMETRICS system appears to have many of the previous problems of the STONEMAN and Army specification, it has some improved features and specifications--e.g.:

In IR-676 (Draft) 13 March 1981
- On page 19: 3.3.7.2 requires the MAPSE to have "features to protect itself from user and system errors."

- On page 21: 3.7.1 requires the KAPSE/Database (presumably for host and target?) to provide "exclusive access..." and "system recovery/reconfiguration..."

- On page 22: 3.7.3 implies good program integration through configuration management.

- On page 24, however, security is not discussed in dealing with the editor and the debugging facilities.

However, in IR-678, the database system appears to be somewhat crude by modern DBMS standards.

(c) DBMS and Dictionary Systems

The rapid growth of the Dictionary System (DS) industry is one proof of the change in modern information systems development methodology. The modern DS is often, however, linked to "its" database management system(s), either through common features or "active interfaces." The common features are frequently found in the architecture: the DS is implemented on a DBMS. There are, of course, "stand alone" DS, but these tend to be the minority. Examples of common systems are Cullinane's IDD/IDMS; Cincom's TOTAL Information Systems Concept; UCC's UCC10/IMS; IBM's Dictionary/IMS, etc. (in fact, essentially every DBMS vendor now has its own tied dictionary). Examples of stand-alone systems include Synergetic's Data Catalogue 2 and MRP's Data Manager.

The active interfaces between DS and DBMS may be in either direction: The DS may receive information from the DBMS or vice versa. The degree of control between DS and DBMS may vary from none to high. Thus a DS may provide the data description for the DBMS schema through one active interface.

The schema may be constrained to be generated only through the dictionary (thus the schema cannot be separately generated and this implies high control), but the control may be non existent, with passive use of the dictionary to aid in schema generation.

One interesting architecture for a KAPSE DBMS/DS interface is one with tailorable controls; i.e., the use of the DBMS as a basis for the DS, and the provision of active interfaces between them, plus the inclusion of a control interface that can be controlled.

(d) Extensible Dictionary/Configuration Entities

The modern Dictionary has extensible features (a discussion of these is presented in Issue Paper # 1 of the appendix: the Issue Papers were generated by the Alpha Omega Group, Inc. for the National Bureau of Standards as part of contract # NB81SBCA0735). We suggest that these can be used to implement early tools for control and implement the configuration entities needed for a good system. Indeed, any other method appears to lose flexibility and also entail unnecessary duplication of implementation effort.

As an example, we have found in some preliminary investigations and initial design specifications, that the use of a relational DBMS (R-DBMS) with a prototype "tabular" dictionary and a triggered control through a relational

calculus interface can provide full dictionary extensibility without significant increase in system size. Moreover, the ability to control the interface between the dictionary and the data promises relatively easy configuration management implementations. Although this material is not yet fully available, we expect to be publishing a preliminary discussion in the next few months and to have an architecture available within the year for a full configuration management and dictionary interface.

Part of the problem of software development, versions, synonyms, and the tracking of software, data, and user entities is in the naming conventions. Some other problems of integrity and security (including auditability) also appear to be managed within this environment of extensibility. These problems are addressed in the other issue papers attached as the appendix.

(e) Extensible Dictionary and ISR Entities

It is our contention that the same features of extensibility that allow the DS to be used as part of configuration management also provide an environment for good classification, indexing, and retrieval of information about reusable software, data, etc.

The provision of a good classification method, we suggest, requires a module able to aid in "faceted classification" methods. These are an intellectually rich rather than a technically provided feature. We believe, however, that good structures exist for software, and we plan to pursue them. However, once the structures have been formulated, it is still necessary to have user friendly interfaces to an ISR support facility. These are only in an early design phase, but we feel that the whole package is vital for future ADA systems viability in the software productivity area.

APPENDIX

The issue papers originally included in this point paper are part of the deliverables of contract #NB81SBCA0735 (National Bureau of Standards). These papers have not yet been released for general distribution. Contact NBS and reference the contract number for further information.

Issue paper titles are:

1. The Dictionary Schema
2. Synonym/Alias
3. Integrity Rules for Adding Relationships to the Dictionary
4. Status, Staging and Version Facilities
5. Audit Requirements
6. Security
7. Interactive Language Characteristics

APSE PORTABILITY ISSUE - PRAGMATIC LIMITATIONS

Herb Willman
(Raytheon)

The General Problem of Software Portability

The general problem of transporting a program from one system to another can be stated as follows:

A program, P, which has compiled correctly on a host, A, and executes correctly on a target, X, is transported to a target, Y, by recompilation on a host, B.

Several possible outcomes of this rehosting/retargeting exist:

(a) P compiles correctly on B and executes correctly on Y, producing results exactly equal to those from execution on X

(b) P compiles correctly on B and executes correctly on Y, producing different results, but less correct, than those from execution on X

(c) P compiles correctly on B and executes correctly on Y, producing different results, but more correct, than those from execution on X

(d) P compiles correctly on B and executes incorrectly on Y

(e) P compiles incorrectly on B

Outcome (a) is the usually desired result: no portability problem exists.

Outcomes (b) and (c) may result when a program is retargeted to a machine which has a different representation for real variables. This might occur when a program is retargeted from a 16 bit word length machine to a 32 bit word length machine, or vice versa. These outcomes may also result from different execution times on Y yielding differing control signal or message transmission timing.

Outcome (d) may occur when the run time environments differ between X and Y. This may result in unacceptably different numerical results or control signal timing. The differing numerical results may occur because of different mathematical libraries being used or because of different representations, or both. The different control signal timing may result from different run time control programs having different priority structures or from execution time differences. Limitations on the features of the language supported by the run-time environment. More on this later.

Outcome (e) may result from different compilers having different built in limitations on the source code which is acceptable to the compilers. Both compilers may be verified compilers. One compiler may compile a program correctly and another may not be able to do so. How this may happen is the focus for the remainder of this paper.

Pragmatic Limitations - An Ada Portability Issue

With the formalization of Ada syntax and the eventual formalization of Ada semantics, one potentially troublesome area that remains is the Ada pragmatics.

The pragmatics of a mechanical language usually refers to its applicability to a user problem domain. It addresses the languages utility for a particular application. In part, this utility may be limited by the compiler system designers. Because of the finiteness of the implementation environment or because of a concern for compiler system operating efficiency, features of the language may be restricted by design. These limitations arise due to the real need by implementors to place a bound on certain language features not otherwise limited by specification or standardization. These restrictions are derived from the finite sizes of records, arrays, lists, stacks, strings, etc. within the compiler system itself and also within the run time environment.

Therefore portability problems may arise because the pragmatics of different compilers may be different. When a program is transported from one computer type to another by recompilation, there may be areas of that program that have to be rewritten due to these differences.

Examples of these differences are:

- Maximum identifier length
- Maximum number of identifiers in a compilation unit
- Maximum number of operators in an expression
- Maximum number of enumeration values in a declaration
- Maximum depth of recursion

These differences may not be limited to the compilers. The pragmatics of the APSEs themselves, especially in the KAPSE/MAPSE interface, may impose similar constraints. For example, the maximum number of compilation units and the maximum number of external references that the link editors can process may be different.

Discussion of the Issue

Appendix A contains a list of items from the Ada Reference Manual (July 1980) which are subject to pragmatic limitation by compiler system implementors, and which therefore may present obstacles to portability. However, it may not be simple to identify the values for each of these items for a particular implementation. Many of these items may share storage area during compilation and the pragmatic limitations may occur with combinations of items. For example, a string containing all of the identifiers in a compilation unit may also contain all of the enumeration values. The maximum number of identifiers then depends on the number of enumeration values, and vice versa.

This difficulty aside, it is still important to know the pragmatic limitations of the relevant compilers when faced with transporting programs from one system to another.

Consider the following hypothetical situation of rehosting a program from target X compiled on host A to target Y compiled on host B. The pragmatic limitations of both compilers for a small set of items is listed in the table below:

Items	Host A/Target X	Host B/Target Y
max identifier length	22	32
max length of strings	256	4096
max dimension of ARRAYs	3	9
max depth of recursion	128	64
max # enumeration values	32	65536
max # PACKAGEs in WITH clause	8	32
max # explicit exceptions	16	128
max # compilation units	128	400
max # statements in comp unit	4095	512
max # active tasks	256	8

It can be readily seen from the comparison above that unless these limitations are known to the user, it is highly likely that transporting a program from X to Y or from Y to X will be a major problem requiring significant redesign, rewriting, and recoding of the rehosted program.

This type of problem is a commonly occurring one during the rehosting of programs which are components of large DoD systems. At Raytheon, we have encountered it often when rehosting JOVIAL/J3 tools from one system to another. Internal tables overflowing is the common symptom for such a problem and, if you are lucky, the compiler detects the overflow and degrades softly with explanation. Occasionally they just crash.

Since portability is a major reason for the development of Ada, we recommend that compiler pragmatics and also APSE/MAPSE/KAPSE pragmatics be recognized as a potential obstacle to portability and we make several recommendations in this regard.

Recommendations for Further Effort

Several recommendations can be made for further effort in this area.

First, the current developments can be reviewed and if possible their pragmatic limitations determined for each of the items in appendix A. Publication of this list would enable the user concerned with portability to select from the list those limitations which represent the common subset. This would allow programs to be compiled successfully under both the Ada Language System (ALS) and Ada Integrated Environment (AIE) compilers. For example, if one compiler system allowed identifiers of maximum length of 22 characters and the other compiler allowed a maximum length of 32 characters, then a user could standardize on 22 character identifiers and be assured of successful compilation on both systems with respect to this item.

Second, the KAPSE Interface Team (KIT) could request that the developers of the ALS and the AIE agree to a common standard for such limitations and thereby remove these obstacles from the two developments. The common standard could be developed by the KIT and the industry/academia support team. If a common standard is not possible or practical at this time, both developers could change those areas which are not yet designed to have common limits. Since the C5 Specifications are not yet completed, this may be practical.

Third, the AJPO could develop a standard or a set of guidelines which would establish the minimum capability of an Ada compiler system with respect to these limitations. If necessary, two standards could be established to allow for the differences in requirements for small micro/mini processor systems that might be found embedded in a missile versus the requirements for a large ground based command/control/communications system. This list of minimum capability would then be the common subset of capabilities to be used when portability is a prime concern. These standards could be validated by the Ada Compiler Validation Suite (ACVS). If multiple guidelines are viewed as "subsetting", then perhaps a single guideline will do.

As an initial guideline, the Ada Reference Manual can be used to derive some minimums. For example, an Ada compiler should be capable of handling identifiers of at least 22 characters in length, since the identifiers for the generic library procedures SHARED_VARIABLE_UPDATE (Sect. 9.11) and UNCHECKED_DEALLOCATION (Sect. 13.10.1) are 22 characters long. But standardization by implication is a weak approach, and in this case is inadequate for a complete treatment.

The Ada Compiler Validation Suite is also a source for implicit minimum capabilities and the ACVS could be modified to "explore" a candidate compiler for its limits.

Fourth, the KIT and the industry/academia support team should investigate the pragmatic limitations of the current KAPSE/MAPSEs under development and determine the extent of the problem which may exist. The KIT could then issue standards or guidelines for this area.

List of Items

Length of Identifiers
Number of Identifiers
Number of TYPE Declarations
Number of SUBTYPEs of a TYPE
Number of Enumeration Values
Dimensionality of ARRAYs
Length of Strings
Number of Record Components
Number of Variant Parts
Number of Declarations
Number of Attributes
Number of Literals
Number of Operators in an Expression
Number of Objects in an Expression
Depth of Nested Parenthesis in an Expression
Number of FUNCTIONs in an Expression
Depth of Nesting of IF statements
Number of ELSIF Alternatives
Number of CASE Alternatives
Depth of Nesting of CASE Statements
Depth of Nesting of LOOP Statements
Number of Declarations in a Block
Number of Statements in sequence_of_statements
Depth of Nesting of Blocks
Length of Labels
Depth of Recursion
Number of Declarations in a Subprogram
Number of Actual Parameters in a Subprogram Declaration
Number of Declarations (visible) in a PACKAGE
Number of Declarations (private) in a PACKAGE
Number of PACKAGE Names in a USE Clause
Number of PACKAGE Names in a WITH Clause

Number of ACCEPTs
Number of ENTRYs
Number of SELECT Alternatives
Number of DELAYs
Number of PRIORITYs
Number of Simultaneously Active TASKs
Number of Explicit EXCEPTIONs
Number of Declarations in a Compilation Unit
Number of Statements in a Compilation Unit
Number of Compilation Units
Number of Library Units

THE ROLE OF THE PERSONAL WORKSTATION IN AN ADA PROGRAM SUPPORT ENVIRONMENT

J. Ruby
(Hughes Aircraft)

Introduction

A significant recent development in the evolution of software engineering support environments is the personal workstation. It is now possible to buy, for example, a 256KB 8086-based workstation with more than 30 megabytes of disk storage for less than $15,000. While this is still far more expensive than a "dumb" CRT terminal, the price is rapidly dropping into a range where it deserves serious consideration as a essential addition to every programmer's tool kit. The advantages of providing such a resource to each software engineer have been well documented in the literature (1, 2, 3) and include the following:

- consistency of response time in interactive sessions
- high bandwidth interaction (particularly important for graphic interfaces)
- constant availability
- independent reconfigurability

Moreover, personal workstations can give software engineers greater control over their development environments, helping them to make more

effective use of their time and improving their productivity. By connecting such powerful resources together over high speed local communications networks, that include shared resources (e.g., specialized data base machines), large software engineering projects can be carried out in a distributed yet configuration-controlled environment. This scenario invites one to view an APSE as a distributed system, with every workstation running a KAPSE along with a significant set of MAPSE tools. It is far from clear, however, exactly how APSE objects could or should be distributed and controlled in such an environment. This paper identifies some of the issues that need to be addressed in designing a personal workstation based distributed APSE and discusses an approach to resolving some of them.

Issues

Among the issues raised by the above notion of a distributed APSE are the following:

- What KAPSE functions must be supported in each workstation? What extensions are needed to handle network communication?
- How should the data base of "objects" for a given project be distributed and shared across workstations and other network resources? How can the integrity of shared objects be maintained without eroding the benefits (e.g., consistent response, high bandwidth communication) of personal workstations?

In addressing the first issue, we note that the "Stoneman" report (5) defines several specific MAPSE functions. As noted in (4), these include Ada run-time support (tasking, exceptions, predefined library units) and tool composition. Since all MAPSE tools must be implemented in Ada, it is clear that complete Ada run-time support must be provided in each workstation. Furthermore, since Ada makes no provision for distributed computation, these KAPSE functions are entirely local. However, tool composition is another matter. It is entirely possible, even likely, that individual workstations will host distinct MAPSE tools, and that some network resources may host special tools shared by many network users. The KAPSE must therefore provide the necessary communication mechanism to enable a tool hosted on one workstation to invoke a tool residing elsewhere in the network (e.g., a query processor hosted on a shared database machine).

In addressing the second issue, it must be noted that the data base is the heart of an APSE system and is a particularly critical element in the distribution problem. In a large scale software project, there may be hundreds of software engineers developing thousands of program modules, all of which must be integrated to form the deliverable system. It is obviously not feasible to divide such a system up perfectly into disjointed pieces so that each programmer can perform isolated development totally within his/her own workstation environment. It appears likely, therefore, that a shared database machine would have to be part of the APSE network for such a project, and that all objects (e.g., requirements specifications, design documents, source and object text, etc.) would ultimately reside in such a specialized node. This immediately

gives rise to a potential resource contention problem that the personal workstation approach is intended to obviate. Some observations about the nature of this problem, and its implications for data base object management, are discussed below.

Configuration Control

A major problem in the development of large software systems is configuration control: how can a team of programmers work cooperatively on overlapping portions of a large software system without stepping on each other? This problem can be addressed in systems which employ a single central data base by supplying appropriate locking mechanisms to guarantee that objects locked by one programmer will be inaccessible to others. Unfortunately, the overhead of such mechanisms further degrades the quality of programmer interaction in a totally shared environment, particularly if the locking is done at a fine level of granularity (e.g., at the record level as proposed in (6)).

However, one may expect a data base of software development objects to exhibit a reasonable degree of partitionability along many different dimensions, as suggested in (7). In particular, it should be possible to impose a partitioning according to responsible individual. Thus, for example, a particular software engineer may control or <u>own</u> a partition of the high level design specification (e.g., a specific subsystem as defined in a collection of structure chart objects); or a programmer might own a distinct collection of modules (including source text, object code, load image, etc.). In principle, such owner-based partitions could serve as the basis for fully distributing the database

across personal workstations. However, such an approach would lead to excessive fragmentation of other logical partitions (e.g., partitions of objects that are derived from one another in migrating through the software life cycle). It would also result in potentially heavy interaction between workstations housing highly interrelated objects, thus negatively impacting the performance of each.

For these reasons, it seems better to postulate the existence of a central data base machine resource that is accessible to all workstations via a high speed local network. In such a configuration, modification rights could be preserved in a variety of ways. One approach that takes advantage of the workstation characteristics is to require that all modifications to data base objects be made first on the owner's workstation. Thus it would be necessary, for example, for each programmer periodically to impound some portion of the data base, making a local copy of it on his workstation. In so doing, he becomes the holder of the primary copy of those objects (until he releases them), and this status is recorded at the central data base processor. Other programmers working on related objects are free only to access the original (now secondary) copy of the impounded objects and in so doing will be informed of their impounded status. However, they are not allowed to impound them for modification (even those they own). Meanwhile, the impounding programmer enjoys high bandwidth interaction with the impounded objects in his personal computing environment. When he has completed his interaction, he returns the (presumably partially modified) objects to the data base machine for consistency checking and update.

Note that this is a very conservative approach to data base integrity preservation: no impounded object, whether owned by the modifier or simply being used as context, may be involved in two concurrent modification sessions. This guards against the erroneous use of non-current context in, for example, testing a change to a module that relies on services provided by a package with a different owner. This level of protection may be unnecessary for non-executable objects (e.g., design structure charts).

As noted earlier, a programmer must be able to impound objects that he does not own. In such cases, the non-owned objects may not be changed by the interaction--if they are, the ensuing attempt to update the central data base must be rejected.

The viability of the above approach is heavily dependent on the aforementioned partitionability of the data base. If the overall software design and project work flow are such that frequent conflicts occur in attempts to impound, much of the value of the personal workstation will be negated. In addition, the number and size of impounded objects needed for a typical interactive session must be small enough to allow rapid copying at local network speeds (10 Mbits/sec), yet large enough to enable a reasonably lengthy local session. Finally, the problem of successfully merging changed objects back into the data base, with appropriate version control, must be studied in the context of various software development object types.

Conclusions

The trend towards personal development environments for software engineers is likely to accelerate as the cost of significant computing power continues to decline and the availability of reliable, high-speed local networks and back-end data base machines becomes more widespread. Significant problems of resource sharing in large software engineering projects need to be solved, however, before the potential of personal workstations can be realized in the form of a distributed APSE.

References

1. Cutz, S., Wasserman, A., and Spier, M.: "Personal Development Systems for the Professional Programmer", Computer, v.14, no. 4 April 1981.

2. Teitleman, W., and Masinter, L.,: "The Interlisp Programming Environment", Computer, v.14, no. 4, April 1981.

3. Wirth, N.,: "Lilith: A Personal Computer for the Software Engineer", Proc 5th Intl Conf on Software Engineering, San Diego, CA., March 1981.

4. Stenning, V., et. al.,: "The Ada Environment: A Perspective", Computer, v.14, no.6, June 1981.

5. Requirements for Ada Programming Support Environments "STONEMAN", U.S. Dept. of Defense, Feb 1980.

6. Design of the Ada Integrated Environment , Texas Instruments Technical Report (Pratt), March 1981.

7. System Specification for the Ada Integrated Environment, Computer Sciences Corp., et. al., March 1981.

ADA PROGRAM SUPPORT ENVIRONMENTS REQUIREMENTS

NOTES ON KERNEL APSE INTERFACE ISSUES

Rob Westerman
(TNO - IBBC, The Netherlands)

Abstract

This paper presents a personal view on Ada Programming Support Environments (APSEs), as stated in the "Stoneman" requirements. In particular it focuses on some areas of "Stoneman" which caught the attention of the author and in his opinion are prime candidates for further investigation and perhaps standardization. Next it gives an idea how the set of interface definitions that must be provided by the KAPSE should be structured. And it also considers the socalled host/target approach which forms the basic idea of "Stoneman".

Prologue

This paper presents a general view on Ada Programming Support Environments as stated in the "Stoneman" requirements (1). In particular it focuses on some areas of "Stoneman" which caught the attention of the author and in his opinion are prime candidates for further investigation and perhaps standardization. It gives an idea how the set of interface definitions that must be provided by the KAPSE should be structured. And it also considers the so-called host/target approach which forms the basic idea of "Stoneman". In such a configuration the APSE is regarded as distributed between the machines.

The following sections of this paper address general considerations, KAPSE design considerations and the area of host/target configurations.

General Considerations

This section presents a view on the Requirements for Ada Program Support Environments as stated in the "Stoneman". It is not directly concerned with an explicit interface issue, which perhaps could have been overlooked nor is it concerned with transportability requirements. It is primarily concerned with pointing out which section of the "Stoneman" should be given more elaborate consideration.

It is today that we are going to make an attempt to specify a standard set of KAPSE interfaces.

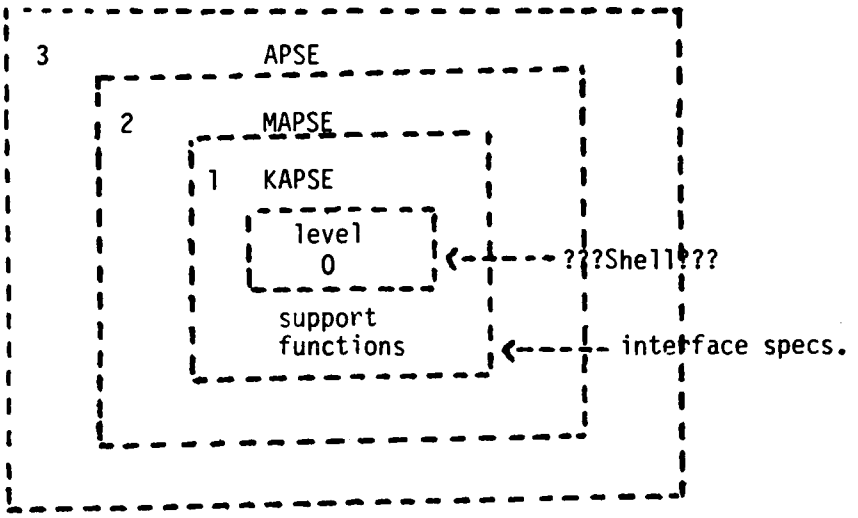
According to "Stoneman":

"2.C.5 In order to achieve the long-term goal of portability of software tools and application systems dependent on them, it is intended that conventions and, eventually, standards be developed at the KAPSE interface level."

This paragraph immediately reminds us of the often painful and time taking procedures from the past which were necessary to agree on a standard for merely a programming language. If these efforts only were successful, but even for a concise, reasonably neat language as Pascal it took far too long to come to a consensus. The reason behind all this of course, besides nature of man, is the overwhelming variety of hardware, where the final programs are going to run on and thus going to be dependent on. And as long as new hardware will appear, these problems will stay.

This brings us to the point of vendor supplied software. In most cases, up to now, the hardware does not come alone. It is always accompanied by some "computing environment" and what's more, this environment has always been built keeping in mind that it has to fit on this particular hardware. It therefore takes maximum advantage of the underlying hardware and thus must always be used in combination with that hardware. These developments have led to a class of very rigid systems, which are in no matter portable or able to communicate with each other.

Still it is encouraging that there are people who think that these problems can be solved. Not only for mere programming languages, but even for Program Support Environments. From the previous it will be obvious that the shell that connects the KAPSE with the hardware is the most important one to the overall picture of an APSE. This point of view is expressed in the following figure.



Structure of an APSE

According to "Stoneman":

"1.E It is convenient to represent an APSE as a structure with a number of layers or levels:

0: Hardware and host software as appropriate.

1: Kernel Ada Program Support Environment (KAPSE), which provides database, communication and run-time support functions to enable the execution of a Ada program and which presents a machine-independent portability interface.

2: Minimal Ada Program Support Environment (MAPSE), which provides a minimal set of tools, written in Ada and supported by the MAPSE, which are both necessary and sufficient for the development and continuing support of Ada programs.

3: Ada Program Support Environments (APSEs), which are constructed by extensions of the MAPSE to provide fuller support of particular applications or methodologies."

KAPSE Design Considerations

This section presents a way of looking at a KAPSE and gives an idea on how the set of interface definitions should be structured.

According to "Stoneman":

"5.E.1 The KAPSE shall implement interface definitions which shall be available to APSE programs."

We can think of a KAPSE as a series of layer functions beginning immediately above the hardware and extending outward. Then it should be obvious that the interface definition effort should start at the layer which is the closest to the hardware. Suppose, that this effort succeeded and that we managed to implement the set of interface definitions of the KAPSE. Let us assume that the not hypothetical case that manufacturer X supplies us with a new exotic device or processor that we want to use. This new hardware must be monitored by the KAPSE. Because this hardware will probably have impact on resource management, it will affect the interface that we just with a lot of effort managed to get into work. Thus in order to keep up with the requirements, which speak of portability of tools and databases we just have to forget about this new hardware.

Of course this is not what we want. What we do want is to define a flexible interface structure that enables us to keep up with these new hardware trends. And still gives us the opportunity to use our existing tools and databases. In other words, the KAPSE interface definition set must be flexible and more or less open-ended, if possible, so that at any time the set of interfaces can be adapted without violating the definition set. And by inference, without disturbing the already existing software.

This attitude is conform to the next paragraph from "Stoneman."

"3.K HARDWARE: an APSE will be designed to exploit, but not demand modern high capacity and high performance host system hardware."

Anyway, the important issue is that we are now in the position to specify interfaces, that must help us to overcome not only our today problems but also our future problems. We have to be aware of the fact that KAPSE interface definitions have to be both ways adaptable. At one hand there is the hardware, ever changing. And at the other hand there are the APSE programs that eventually have to use this hardware and thus have to communicate with the standard virtual interface that is supported by the KAPSE.

Host/Target Approach

This section considers the assumption that the APSE in a host/target approach is regarded as distributed between host and target.

According to "Stoneman":

"2.B.2 An APSE adopts a host/target approach to software construction. That is, a program which will execute in an embedded target computer is developed on a host computer which offers extensive support facilities.

Except where explicitly stated otherwise, this document refers to an APSE system running on a host machine and supporting development of a program for an embedded target machine."

Given such a distributed system, where does the environment reside? No doubt, parts will be local at the target and parts will be at the host. This will depend on the size of the target and on the way the target is going to be used. All sorts of configurations are possible. The host/target configuration at which we want to take a closer look, is the one where the socalled testsupervisor is not merely an Ada program, but more or less a MAPSE in itself. Of course this will be the case if the target supervisor has to be able to perform some general tasks like:

- o supporting I/O
- o managing Files
- o managing System Resources

What impact does this kind of configuration have on the term distributed? Does this mean that there is also an environment on the target where the target software under runs? Does this mean that the KAPSE also is regarded as distributed? This is an area that is not enough covered by "Stoneman" and surely needs more elaborate consideration.

Epilogue

Although up to now standardization only for a few programming languages succeeded, it is encouraging to see that there is hope even for standardization of (specific parts of) APSEs. It is almost trivial that most efforts should be put into the lowest level of the Kapse in order to get eventually a flexible implementation of an APSE. The host/target approach which forms the basic idea of "Stoneman" must be given careful consideration, because this leads to distributed APSEs. Especially when we consider the target software as being a MAPSE, we feel that this area is not enough addressed by "Stoneman".

References

(1) "Stoneman", Requirements for the Ada Programming Support Environments, J. N. Buxton and V. Stenning eds., February 1980, U.S. Department of Defense.

KAPSE STANDARDIZATION: A FIRST STEP

D. E. Wrege
(Control Data Corporation)

The major goals of the DoD common high-order language effort are to:

- address the problem of life-cycle software costs, and
- improve the quality (reliability) of software

for embedded computer applications. DoD recognized that these objectives would not be met by the Ada Language alone, but by a comprehensive, integrated programming environment.

The need for such an environment is not a newfound revelation on the part of the DoD; in fact, the software engineering community has recognized the need for such environments for some time. The problem is that 1) comprehensive, integrated environments have not been made commonly available by the commercial community (largely because they do not have the same need for them that the DoD does), and 2) such environments are too costly for contractors to develop in order to support single or even multiple application efforts. Thus, embedded computer software has typically been developed in austere programming environments, with other life-cycle phases being regarded as "someone else's problem."

It thus became apparent that in oder to address the DoD goals of cost-effective development of high quality software, one must

- promote the development of portable software, and
- promote the development of portable software tools.

In short, one must have a portable environment. Portability is, therefore, a keystone for cost-effective software development. In recognition of this fact, STONEMAN required portable tools:

> "... a further requirement is portability both of APSE tools between ... APSE's hosted on different machines and of complete toolsets." [STONEMAN 2.B.10]

Portability makes it cost-effective to build high quality tools in sufficient number to address the entire software lifecycle, since tools need only be developed once, but can then support an unlimited number of application efforts.

Portability requires the definition of some framework in which tools execute. Recognizing this requirement, STONEMAN defined three distinct layers in the environment: the Kernal Ada Programming Support Environment (KAPSE); the Minimal APSE (MAPSE); and the APSE. The KAPSE is central both to tool portability and to rehosting the entire environment.

"The purpose of the KAPSE is to allow portable tools to be produced and to support a basic machine-independent user interface to an APSE." [STONEMAN 2.B.11]

Any program which both uses only standard Ada constructs and makes external reference only to the KAPSE is portable, without modification, to any host offering the same KAPSE. (Note: Definitions of Data types that are involved in basic intertool interfaces must be considered part of the KAPSE definition.)

Thus, a Standardized KAPSE Interface is critical to the development of Portable tools. Stoneman indeed anticipated a standard KAPSE interface

"... the external specifications for the KAPSE will be FIXED." [STONEMAN 2.B.13]

but, unfortunately, only expected a standard KAPSE interface to emerge:

"... it is expected that one such MAPSE, and more particularly, its basic kernel or KAPSE will achieve a sufficiently wide measure of acceptance for it to become a de facto convention and, eventually, for the KAPSE specifications to be considered for standardization within DoD." [STONEMAN 2.C.9]

The Current Situation

The government is currently funding the development of two APSE's: the ALS (Army sponsor/Softech contractor) and the AIE (Air Force sponsor/Intermetrics contractor). These environments will have different KAPSE definitions with attendant non-portability of tools between them. Unquestionably, both of these environments will be useful and will satisfy many of the DoD objectives. It is unfair to criticize the involved participants for failing to have a standard KAPSE: neither has the authority to define one; neither was tasked to define one.

The Ada effort has captured the imagination of the software community and enjoyed popular support as a result of the unified position that the government has taken in its development. The development of the APSE concept has, likewise, elicited widespread participation from government, industry, and academia throughout the STONEMAN process. It is possible that much of this momentum could be diffused by the mere existance of multiple DoD-support APSE's. Indeed, perhaps the greatest threat to the emergence of a "de facto" standard KAPSE, as anticipated by STONEMAN, could be a weak DoD position regarding environments. Consider, for example, what would probably have resulted had there been different Ada dialects for airborn and land-based systems. Independent implementors would surely have developed other subsets/supersets for their "unique" problems: communication, signal processing, networking, etc. The probability of a "de facto" standard KAPSE evolving is almost impossible, given multiple APSE's within DoD.

As an example, Control Data Corporation is currently undertaking the internal development of an APSE. Immediately a decision must be made: do we base our development on the Army APSE or the Air Force APSE? Which do we do most business with? How compatible do we need to be? Shouldn't we wait to see if one or the other emerges as preferred? What is the Navy going to do? ....

It has been repeatedly stated that success of the DoD language effort depends on the widespread acceptance and use of Ada in the entire defense community. Yet management, at least within CDC, percieves a large risk associated with committing internal funds for development of an APSE compatible with "the wrong" DoD-sponsored APSE. Much of the leverage that DoD could have had is lost.

It is CDC's position that THE GOVERNMENT MUST WORK FOR A STANDARD KAPSE and have that as a publicly announced goal of the Ada initiative.

But ... How can a standard KAPSE be defined before we know what is required by the APSE? Isn't it best to develop some environments and see what is required of the KAPSE? We have much less experience with environments than with languages. Development of multiple APSE's will minimize DoD's risk. ... These are very real concerns.

A Standard KAPSE?

Although we don't have a wealth of experience with programming environments, operating systems are fairly well understood. A KAPSE is essentially an operating system that also provides a host-independent interface for the rest of the environment. Standardizing

on such a transparent operating system should not jeopardize the outer layers of the environment. If the KAPSE is to provide the required environment rehostability, however, it must not depend on esoteric features of any particular host operating system.

Consider a KAPSE hierarchically organized as a two-layer system with a clearly defined interface between them. The system can be described in terms of two components:

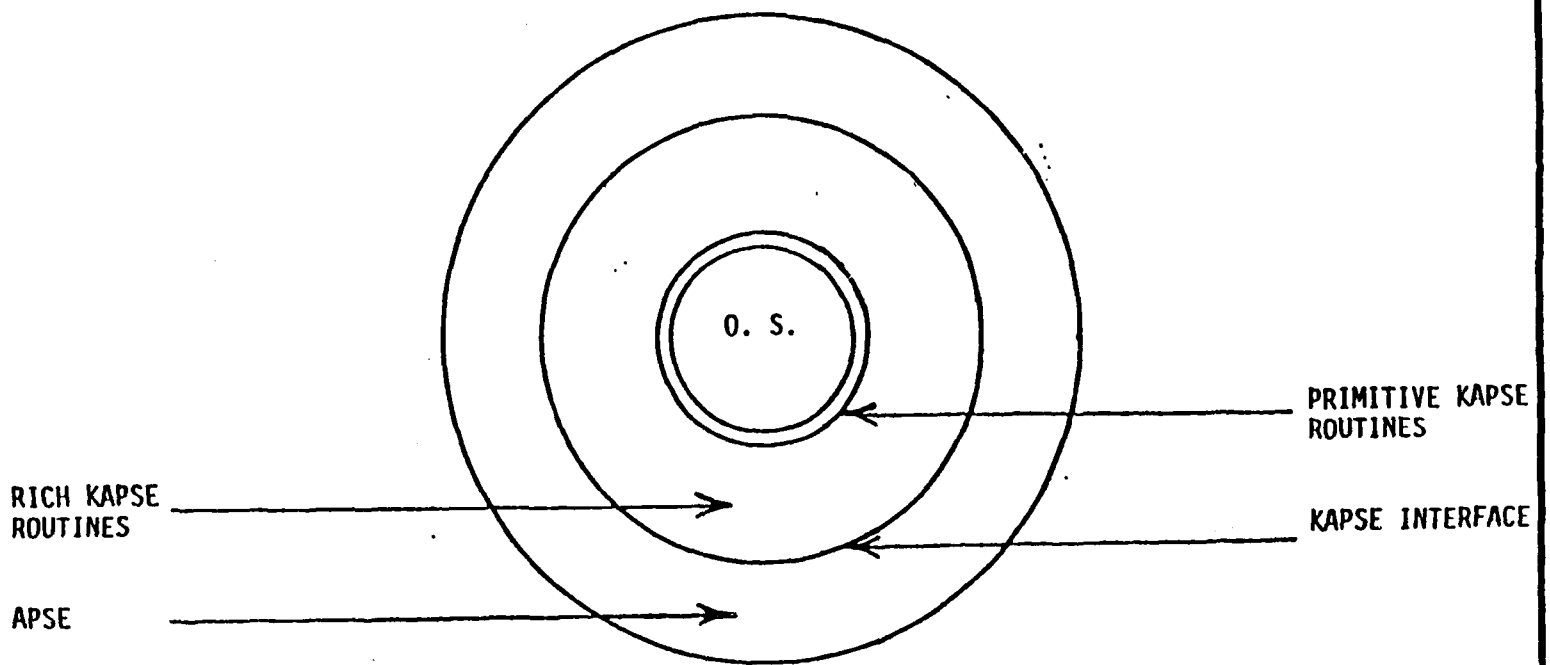
1) An outer shell of "rich" routines that utilize a set of inner, "primitive" routines and that are written in Ada.
2) An inner shell of "primitive" routines that incorporate necessary knowledge of the host operating system. These routines are termed "primitive" since each performs a basic function or service that is common to virtually every operating system. These routines must be Ada-callable, but their implementation language need not be controlled further.

This conceptualization immediately generates two basic questions: 1) What primitive rountines might be needed and 2) Should the rich routines be included "inside" the KAPSE at all?

Some of the basic KAPSE functions which can readily be identified are:

1) Ada virtual machine support - (RTL).
2. Process control - (login, suspend, interrupt, continue, etc.).
3. Program invocation - (wait, no-wait, parameter passing).

Two-Layered KAPSE



CONTROL DATA CORPORATION

AA6133 Rev. 11/77 *79 CDC

PRINTED IN U.S.A.

4. Resource/peripheral allocation and management.

5. Privilege and protection control.

6. Data-base operations, file management, and directory management.

Most of these KAPSE functions can readily be described in terms of "generic" operating system services. Process control, program invocation and resource/peripheral management, for example, are particularly amenable to definition in terms of such operating system "primitives." It is not surprising, therefore, that both the ALS and AIE have taken similar approaches to process control and program invocation mechanisms. Given that tools are written in Ada, implementing Ada virtual machine support functions should not hamper portability either, since the semantics of full Ada will exist on all environments.

Data-base functions, however, do pose a serious problem. These services are not easily expressible in terms of generic operating system functions, since file and directory handling is the aspect of operating systems that varies most from one system to another. It is hardly coincidental that the greatest divergence between the ALS and AIE may be found in their data-base operations. The design of our hypothetical two-layer KAPSE must suggest some solution to this problem.

The rationale for an outer layer of rich KAPSE routines is two-fold. First, providing these rich routines would allow a tool to make a single subroutine call to accomplish complex tasks. The rich KAPSE routines could then assume the responsibility for making a battery of

related calls to KAPSE primitives. This organization would facilitate tool simplicity and adhere to a desirable strategy of information-hiding.

Secondly, the outer layer could contain the data-base routines which are not expressible as generic KAPSE primitives. The "partial portability" of the outer KAPSE routines is a consequence of these data-base operations' host-specificity.

Since the outer routines are to be written in Ada, they will be portable as code. Host-independent routines, which serve to effect a collection of related calls to KAPSE primitives, will, therefore, be rehostable. The host-dependent routines, which deal with file organization particulars or which preserve the integrity of the data-base by handling such problems as simultaneous user access, can still be written in Ada for comprehensibility. Such host-dependent procedures should be carefully isolated and clearly marked.

A layered KAPSE approach might at least serve as a first step toward KAPSE standardization. Distinct environments, such as the ALS and AIE, would still exist, but they would be more easily hosted on a single machine. This increased availability, with the resultant opportunities for environment comparison, could encourage the envisioned emergence of a de facto standard KAPSE.

POSITION PAPER

Harrison R. Morse
(Frey Federal Systems)

Goal

The goal of this paper is to address two areas of individual concern to me as a potential user of Ada, and a potential developer of environmental support (operating systems) for Ada on embedded systems, namely:

1. Transportable Environment for real-time multi-tasking Ada-based systems.

2. Definition of techniques and facilities for capturing exception and performance data needed to support system checkout, interface verification, system performance evaluations, and provide default responses to exception conditions.

The first of these requires no additional explanation at this point.

The second derives from both recent and long-term experiences in the development of systems software to support the development of applications programs in real-time environments.

My desire to raise this latter issue in a forum such as this is three fold:

1. Design of such support tools is often left until a system's well into checkout, resulting in ad hoc approaches that can be charactized by "too little, too late".

2. Techniques developed for one system are seldom propogated.

3. Exceptions are handled on an "as they occur in the code development" basis, with no common control point for either presentation control (critical in debugging and development) or system response control (critical in operation).

As discussed below, I propose to attempt developing standard techniques for providing access to such information.

Unfortunately, if such tools and techniques are developed at all, they are ad hoc, too little, too late, and not general purpose.

The activity proposed is to address the development of standard techniques for embedding "connectors" to:

- Data Collection and recording (on-line and off-line) tools
- "Action" routines
- Interface verification routines
- Trace and flow recording mechanisms
- Debug controls
- Exception and error handling

The initial goal of this effort will be to address the development of a general technique within the APSE, and provide directions toward defining specific mechanism on a per system basis.

Notes

My overview is that "connectors" are always embedded in the systems to avoid code changes after late stages of testing.

The "action" routines may be linked or not (replaced by null stubs), and the invocation of any routines may be controlled dynamically, by system parameters or flags.

Disposition of results produced by invoked routines should be dynamically controllable, for example: written to disc, displayed, or ignored.

Background Notes

The following notes outline some systems and their critical design goals which form some of my views.

As most of us, many of my design goals or decisions reflect overreactions to my most recent scars, normally resulting from support systems which (by design or implementation deficiencies) have impeded rather that enhanced the development process. Hopefully, as I get older, the net result of these scars will result in a more balanced reaction in design approaches to support system development.

1. TOPS-10 (Role = designer/implementation - DEC, 1962, 1963)

   - Configuration independent support environment for development Tools
   - Interactive access/time-sharing as a primitive feature (multi processing assumed)
   - Device independent I/O access

2. Files-11 (Role = Designer - DEC 1971)

- Virtual disc (device independent)
- Crash proof structures (implication on update techniques)
- Potential for efficiency - File access: asymptotic to 1 disc access per disc access
- Transportable techniques
- Full multi-access files supported
- Data management a higher level — always

3. RSX - Family (Role: User, extender of capabilities)

RSX-11A, RSX-11D, RSX-11M, VAX/VMS

- Q-Oriented, especially for I/O
- Priority task scheduling
- Asynchronous control mechanisms
- Underlying mechanisms efficient by design (especially VMS)

4. Cybos (Like Forth) - Virtual machine (16 bit, stack oriented) with fully transportable code and data bases - for application development/support environment

To transport, the following must be coded at the machine level:

- I/O drivers TTY, disc, printer, magnetic tape
- Scheduler
- Primitives (in machine language or micro-code)

The rest transports directly. The support system ("kernel"?) can be implemented on a bare machine (DG Eclipse, PDP-11, IBM Series I) or on a host operating system (VAX, IBM 4331).

KAPSE INTERFACE STANDARDS

Thomas A. Standish
(University of California at Irvine)

Introduction

Ada beckons us with the promise that a medium of exchange will be established enabling us to exchange programs and the data they operate on.

But even if the Ada language is given a precisely defined, standard meaning, and even if Ada compilers are certified to meet that standard, it does not follow that we can exchange programs and data. The reason is clear --- programs are written to hook up to devices, to call on services, and to exchange data and control with the surroundings in which they execute. Only when there is invariance between a source S and a target T of Ada calling forms, interface conventions, data formats, and control exchange agreements, can we expect to transfer an Ada program from S to T and have it run on T with the same effect it had on S.

Suppose we agree on the meaning of three technical terms: transportability, interoperability, and reusability. Transportability means we can move Ada programs from a source to a target, enabling, for instance, the sharing of tools. Interoperability means we can move data from one KAPSE database to another, enabling the transfer of data from a source to a target. Reusability means we can take Ada software in one application and use it again in a new application.

Of these three capabilities, let's suppose we are shooting only for transportability and interoperability and not for reusability. In the remainder of this position paper we discuss, first, transportability and second, interoperability.

Transportability

See if you agree with the following proposition:

> Proposition: A sufficient condition for an Ada program P to transfer from a source machine S to a target machine T and to have identical behavior on both machines, is that each exchange of control or data of P with S and T have identical behavior.

If you agree with this proposition, then you might be persuaded that the following technical approach to transportability merits investigation:

1. Suppose we could identify a (perhaps large) set S of ways in which programs exchange control and data with contemporary host operating systems.

2. Then, suppose we could define a class C of operating system services (together with their data and control conventions) which could be used to express the exchanges in S.

3. Finally, suppose we could express the services in C as an Ada package specification P, (perhaps supplemented by representation specifications in the accompanying package body for any concrete implementation).

What we might then be able to do is to write Ada programs calling on "standard" operating system services using calls in P. Such programs should transport without loss of behavioral equivalence to any new machine that supplies the standard meanings (i.e., behaviors) for the services in P.

In other words, if we can define a set of KAPSE virtual operating system services expressed as an Ada package (with rigorous, precise meanings), any Ada program written to call on those services will transport without loss of behavioral equivalence.

(Note: This appears to agree with Buxton's thinking in section 5.F.3 of the STONEMAN where promises are made regarding STONEMAN Appendices giving examples of Ada package specifications. Does anybody know what the progress is on these Appendices?)

Here, then is a recommendation for an activity that could result in a "strawman" package of KAPSE virtual operating system services. We could appoint a handful of subcommittees each to go off and study a separate operating system and to express its operating system services as an Ada package.

I have in mind here candidates such as OS/370, Tops-20, Unix, or your favorites (whatever those may be). In addition to the things you can do at the Command Language Interface (CLI) level, we should also include things reachable via standard operating system service calls at the assembly level. For example, in Tops-20, you can get file directories and dates at the CLI-level, and at the system call (or JSYS) level, you can output a character to a device, open a file for input on a channel, and so forth.

Then there should be an exercise in comparison and synthesis. When we compare the Ada packages resulting from the first step of the suggested exercise, we might be able to identify a common core of "functionality" that all the candidates share, and we may be able to express a new Ada package that "synthesizes" the capabilities revealed in the separate pieces and that "covers" all the functionality revealed. We may be able to do this in a manner that doesn't exhibit bias or favoritism toward a particular manufacturer's way of thinking about the problem.

If we can pull this off, we will have made a healthy stride toward the achievement of our goal of transportability.

I suspect we will never be able to swallow up all the capabilities that need to be expressed to guarantee transportability with this "Ada package" approach --- after all, there are an unbounded set of device-types, database element formats, and interface conventions to deal with. Nonetheless, this approach offers a way of producing a candidate KAPSE virtual operating system interface that will have a great deal of reach and expressivity, and which will, if used, enable the transportation of a vast variety of tools. Whether it can be done without bias, and if done without bias whether it will enable tools to run efficiently remains an open question which we will have to settle once the first step is taken --- and the result is by no means clear. Maybe we are aspiring to pose and answer another "UNCOL" question.

But we can never study effectively whether such a question has an acceptable answer until we perform the hard work of coming up with some detailed candidate interface packages and compare them with an eye toward a synthesis.

Recommendation: Let's roll up our sleeves and get on with it.

Interoperability

The interoperability of databases poses a severe challenge. I'm not at all confident it is possible in general. The number of database formats, file organization principles, views, and query methods is terrifyingly broad.

Maybe we would make progress by trying hard to define a limited yet approachable goal. If we spend a lot of effort defining what we are not going to attempt to do, we may be able to converge on something achievable and useful.

File transfer protocol and ArpaNet mail protocol are two examples of working data exchange agreements that are of proven utility and demonstrable power. Maybe we could start by acknowledging support of these two protocols in the standard virtual KAPSE.

Maybe we could add to these, a capability for exchanging files of Ada data types.

If we had just these three capabilities, we could go far.

Correct me if I'm wrong, but could we not then program up the "objects" in STONEMAN, together with their required and optional attributes, as abstract data types in Ada? Could we not then meet STONEMAN's requirements for support of "objects", "versions", and "configurations" by writing up a "standard" package whose interoperability would rest on the viability of the basic three capabilities mentioned above?

One hesitates to plunge into the world of data description languages, given the track record of accomplishments so far in that field. By cleaving tightly to Ada data type descriptions and some data interchange protocol for shipping containers full of Ada data values, we may be able to avoid perishing in the viper pit of general data description problems.

Is it our job to specify, for example, the abstract data types for DIANA? Or, by contrast, should we make it possible for an Ada package defining DIANA data to run on any machine that supports a "virtual KAPSE interface" and to ship bucketfuls of DIANA trees back and forth using a general exchange protocol for containers of Ada data values?

It might be challenging enough to try to ship containers full of Ada data values. What do we do for shipping Ada access values implemented in heaps, for instance? It seems to me that we need some way of representing "pointers" and their referents that can be shipped in a linear stream (such as a byte stream). Perhaps we can use some special method of labelling access objects with "symbolic addresses", and perhaps we can define ways of translating back and forth from systems of Ada access values and systems of symbolically addressed value representations.

ON THE REQUIREMENTS FOR A MAPSE COMMAND LANGUAGE

Reino Kurki-Suonio and Pekka Lahtinen
(Oy Softplan Ab, Finland)

Abstract

As a compiler language with static bindings Ada requires some kind of a command level for initiating programs and for expressing their execution time bindings. This situation will lead to different kinds of two-level systems where the dynamic layers may vary from conventional command languages to general-purpose interpretive systems. This paper deals with the importance of standardizing the MAPSE Command Language (MCL) for integrated Ada environments, and of a careful review of requirements before such standardization. Two particular aspects are emphasized: a sufficiently powerful MCL with general-purpose capabilities will also be used as an interpretive language in its own right, and it should allow efficient utilization of intelligent terminals connected to the main computer by a local bus.

Introduction

Ada and its programming support environments (APSE) are expected to have strong influence on the design of future computing systems and their user interfaces. Systems are already built where Ada is the main programming language, and where the operating system is designed to provide the necessary support for the kernel of APSE (KAPSE) and for Ada programs in execution. The functional possibilities of a command language (MCL) for a minimal APSE (MAPSE) are crucial for the design of these systems and their Ada integrated environments (AIE).

Designed for efficient compilation with static binding, it is doubtful whether Ada itself is a suitable basis for a command language for initiating Ada programs and for expressing their execution time bindings. This situation is likely to lead to

a variety of two-level AIEs, where the dynamic layers may vary from simple command languages to general-purpose interpretive languages.

The Stoneman requirements for APSEs /1/ leave a lot of freedom for the level of an MCL. The AIE proposal by Intermetrics and COMPASS /2,3/ adopt an ambitious approach where MCL is another programming language with general-purpose capabilities and designed for interpretive execution. Its interface to Ada includes the convenient mechanism of "piping", which, in fact, extends the control structures of Ada with coroutining.

Together, Ada and MCL form a two-level language with similar but unidentical structures on both levels. Both levels have variables, for instance, but their identifiers, types and declarations are different. Since the distinction between the two levels lies mainly in implementation techniques, their separation contradicts those principles that were followed in Ada, when separate language levels for macro expansion and conditional compilation were rejected. Unfortunately, some degree of contradiction with these principles seems unavoidable, unless one is content with a very elementary MCL.

Need for Standardization

There are several reasons that call for standardization of MCL and its interfaces to Ada. Firstly, the run-time support in KAPSE needs to support such MCL extensions to Ada as pipes. Secondly, while Ada programs in execution may also issue MCL commands, transportability of Ada tools requires such standardization.

Thirdly, we consider it most likely that the use of a powerful MCL will not be limited to the typical use of conventional command languages. While Ada will be a language

for specialists, a well-designed MCL would become the first and most important programming language for large groups of beginners and ordinary users of computers. Designed for effective utilization and combination of existing programs and tools, it would allow one to work on a higher level than conventional programming languages or Ada. As such it could be designed to give Ada an essential enhancement for managing complex software systems. With its general-purpose capabilities it might also, as a byproduct, replace interpretive implementations of simple independent languages like Basic.

In fact, these views should lead from an Ada-centered approach to an MCL-centered viewpoint. Instead of considering MCL as an extra layer for initiating programs, one should consider it as the main language for communicating with the computer. According to this view, the role of Ada is to provide efficient means for the specialist to implement the basic modules to be utilized in MCL.

This paper is not intended as criticism against the design of /3/. Rather than trying to polish its details we wish, however, to draw attention to the general MCL requirements, which are now easier to discuss with a detailed design in mind. In view of the above considerations it is urged that the importance and implications of embedding MCL in an interpretive language with general-purpose capabilities be carefully analyzed, and that detailed requirements for a standard MCL be formulated and reviewed before adopting a particular solution. Rather than trying to give a list of such requirements, we try to point out a few issues that should be paid special attention.

Types and Operations

The typing of MCL variables is one of the central issues. A single universal type with a natural external representation and a natural correspondence to some Ada type is probably what is needed. The flexible strings of /3/ provide an otherwise obvious choice, but the lack of similar flexibility in Ada strings causes problems for out and in out parameters. Interface to Ada requires exact specification of how MCL data objects are to be understood in terms of Ada.

Although different kinds of interactive systems can be implemented in Ada, the existence of an interactice MCL with general-purpose capabilities will probably induce various extensions of it with more powerful data structures and operations. For the user such systems would provide the advantage of having a single language for commanding the computer system, utilizing Ada library, and performing non-trivial interactive computations. As a universal data type strings can easily be used to represent different kinds of data structures. A suitable interface to Ada would make it possible to write Ada programs for extending MCL to handle such structures; otherwise such extensions would not be transportable.

Syntactically the interface should at least include ordinary operator notations. The standard meaning of MCL arithmetic operators could then also be given in terms of Ada functions, which could be overridden by some data structure operations if one so wishes.

For the efficiency of implementation it might make a difference, whether an Ada module is intended to be used as an MCL extension or not. In particular, when using programmable terminals which are connected to the main computer by a local bus, one might wish the interactive

processing of MCL and its extensions to run in the terminal. Special pragmas could be used to inform the system about such matters.

Building Systems of Programs

Conceptually the most important feature of the proposal /3/ is probably its pipe mechanism, which allows the standard output of one program to be directed as standard input to another. Experience with Unix , from where it is taken, has made this mechanism the major dynamic way of combining programs together. As such it complements nicely the static ways of combining program modules in Ada.

Although the inclusion of pipes is a major step forward in command languages, it is not at all clear that they should be adopted in MCL without further development or refinement. Careful analysis of variations and generalizations of pipes is recommended in order to arrive at appropriate requirements.

It should be noted that pipes allow only relatively simple possibilities for communication between programs. Even if rendezvous mechanism and similar complex possibilities are out of question, there is a wide spectrum of possibilities for various kinds of dataflow or communication networks between programs. Especially in distributed computer systems, of which systems with intelligent terminals are special cases, such networks might provide useful ways of putting complex program systems together.

Using Interactive Programs Non-Interactively

The standard output of a program may consist of different kinds of information, e.g. prompts, advice, error messages, and results. In particular this is true of programs that have been designed for interactive use.

Problems may easily arise, when such programs are used to communicate with other programs. It seems that it should be possible to consider "standard output" as a merge of several distinct output streams which can be individually redirected or piped.

Similar problems arise also in connection with the MCL command processor itself, when it is used to interpret scripts non-interactively. A particular problem of this kind is caused by exceptions that propagate to the command processor. The proposed solution of providing MCL scripts the possibility of testing the exit status does not seem to be sufficiently convenient for the user. Some kind of an exception mechanism might, therefore, be needed also in MCL.

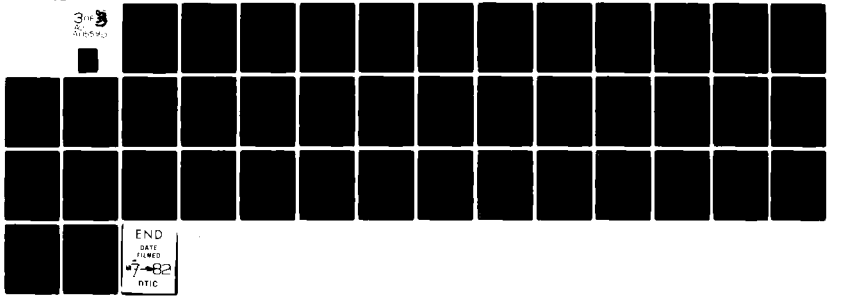
References

1 Requirements for Ada Programming Support Environments, "Stoneman", Department of Defense, February 1980.

2 Ada Integrated Environment, Design Rationale, Technical Report (Interim) IR-684, Intermetrics, March 1981.

3 Computer Program Development Specifications for Ada Integrated Environment: MAPSE Command Processor Type B5, IR-679 (Draft), Intermetrics, March 1981.

AD-A115 590 NAVAL OCEAN SYSTEMS CENTER SAN DIEGO CA F/G 9/2
KERNEL ADA PROGRAMMING SUPPORT ENVIRONMENT (KAPSE) INTERFACE TE--ETC(U)
APR 82 P A OBERNDORF
UNCLASSIFIED NOSC/TD-509- NL



END
DATE FILMED
7-82
DTIC

ADA I/O INTERFACE SPECIFICATION

Stuart C. Schaffner
(Massachusetts Computer Associates)

Introduction

In addition to the basic facilities in the language itself, Ada specifies packages to support I-O, especially sequential text I-O. A user of any Ada language system should be able to use these packages like any other package. It is the responsibility of the purveyors of any Ada language system to correctly support these packages for the I-O devices offered on that system. Some of this responsibility may devolve onto suppliers of I-O devices who may be required to support some or all of the package functionality directly. Thus the functionality of the Ada standard I-O package is of critical importance to both the user of an Ada language system and to suppliers of peripherals for Ada language systems.

Ada provides three levels of I-O package. The middle level is INPUT_OUTPUT, which allows files to be written and read. This supports the concept that a "file" can be an I-O device such as a CRT terminal or a line printer.. The abstraction that an input device is a read-only sequential file and an output device is a write-only sequential file is well-accepted and useful in many circumstances. It leaves control of the device almost completely up to the Ada language system.

The package TEXT_IO is built on the package INPUT_OUTPUT and provides additional control over line lengths and suchlike. This also introduces some potential anomalies if commands from the INPUT_OUTPUT level are called concurrently with the sending of control characters to the TEXT_IO level.

If the TEXT_IO and INPUT_OUTPUT packages do not provide enough control over the device then Ada provides a package LOW_LEVEL_IO, which allows raw control characters to be sent and received. Needless to say, if calls to this level are intermixed with calls to higher levels then anomalies are possible. Because raw characters are sent and received, the resulting actions are completely device-specific.

We feel that another level is needed between LOW_LEVEL_IO and INPUT_OUTPUT. This new level, IO_DEVICES, will expose a substantial number of device characteristics to program control but will standardize the method of control and the meaning of the characteristics. IO_DEVICES will define a number of "exemplary devices" which have clearly defined properties. A manufacturer of I-O devices would then have to produce hardware or software which could map a real I-O device onto one or more exemplary devices. If the exemplary devices included in IO_DEVICES are chosen with sufficient care then this mapping will be quite straightforward and should avoid most anomalies caused by different mappings of the same exemplary device.

IO_DEVICES will also provide better control over the ASCII character set. ASCII contains not only the 95 graphic characters but also 33 control characters. Some of these, such as CR and FF, clearly should be embedded in the character stream. Many others represent device control actions that clearly should not be embedded.

The exemplary devices supported by IO_DEVICES have clearly-defined capabilities. At least the 95 graphic characters and some subset of the 33 control characters will map into well-defined actions on the part of the device. Other device actions will be available only through procedure calls. IO_DEVICES will perform the functional equivalent of a CONSTRAINT_CHECK operation in ASCII to ensure that unsupported characters are not used.

Exemplary Devices

The package IO_DEVICES supports a number of exemplary devices. Each exemplary device represents a class of real devices with similar characteristics. The exemplary devices are specific enough to keep the mapping to real devices straightforward, but at the same time are general enough to keep the number of exemplary devices down to a reasonable level.

One method which is used to keep the number of exemplary devices down is to give such devices optional attributes which can be enabled or disabled. IO_DEVICES defines a data type "capability", as follows:

```
type capability is (enabled, disabled, always_enabled,
  always_disabled);
```

This type allows user programs to negotiate with IO_DEVICES concerning the precise status of a particular device. This negotiation is accomplished through calls on three routines: read, request_change, and force_change. The always_enabled and always_disabled states are "stuck"; attempts to change them will have no effect. Attempts to force them will raise an exception.

A program which is prepared to take advantage of an optional device attribute but which can still function without it can attempt to enable the attribute and check to see if the attempt succeeded. In contrast, a program which is critically dependent on an optional attribute should attempt to force it. This will cause an exception to occur if that program is ever run on incompatible hardware.

It would be better to detect such incompatibilities at compile time or link time rather than at run time. We are currently attempting to find a mechanism which will do this.

Character Sets

One of the purposes of the package IO_DEVICES is to provide a precise definition of the character set for each exemplary device. The basic data type is CHARACTER, defined in package STANDARD. For any given exemplary device, elements of CHARACTER fall into the following classes:

1. basic
2. transformable
3. format effectors
4. device commands
5. nullable characters
6. illegal characters

Associated with each exemplary device is a routine CHECK_TEXT which examines text and raises an exception if any character is a device command or an illegal character. Device driven routines apply CHECK_TEXT to any data passing between the user program and IO_DEVICES. Structurally, CHECK_TEXT is serving the function of CONSTRAINT_CHECK. It cannot be done by CONSTRAINT_CHECK because Ada does not allow subtypes which are arbitrary subsets of enumeration types, but instead allows only subranges.

APPENDICES

A. KAPSE Interface Team Members
B. Industry/Academia Team Memebers
C. APSE Generic Definitions
D. Initial KAPSE Interface Categories
E. KAPSE Category Worksheets

Appendix A

KIT Name list

BALDWIN, Rich	U.S. Army (CECOM)
CASTOR, Jinny	U.S. Air Force (AFWAL/AAAF)
DUDASH, Ed	Naval Surface Weapons Center
HART, Hal	TRW
HOUSE, Ron	Naval Underwater Systems Center
JOHNSON, Doug	Texas Instruments, Inc.
JOHNSTON, Larry	Naval Air Development Center
KRAMER, Jack	Ada Joint Program Office
LINDLEY, Larry	Naval Avionic Center
LOPER, Warren	Naval Ocean Systems Center
MILTON, Donn	Computer Science Corporation
OBERNDORF, Tricia	Naval Ocean Systems Center
OLDHAM, John	TRW
PEELE, Shirley	Fleet Combat Direction Systems Support Activity - Dam Neck
PURRIER, Lee	Fleet Combat Direction System Support Activity - San Diego
WALD, Elizabeth	Naval Research Laboratory
WALTRIP, Chuck	John Hopkins University Applied Physics Laboratory
WHITE, Doug	U.S. Air Force (RADC/COES)
WOLFE, Marty	U.S. Army (CECOM)

Appendix B

Industry/Academia Team

Industry/Academia Meeting 17,18,19 February 1982

CORNHILL, Dennis	Honeywell
COX, Fred	Georgia Institute of Technology
FELLOWS, Jon	Systems Development Corporation
FISCHER, Herman	Litton Data Systems
FREEDMAN, Roy	Hazeltine Corporation
GARGARO, Anthony	Computer Sciences Corporation
GLASEMAN, Steve	Teledyne Systems Corporation
GRIESHEIMER, Eric	McDonnel Douglas Astronautics
JOHNSON, Ron	Boeing Aerospace Corporation
KERNER, Judy	Norden Systems
KOTLER, Reed	Lockheed Missiles and Space
KRAMER, Jack	Ada Joint Program Office
LAMB, Eli	Bell Labs
LINDQUIST, Tim	Virginia Institute of Technology
LOCKE, Doug	IBM
LOPER, Warren	Naval Ocean Systems Center
LYONS, Tim	Software Sciences Ltd (UK)
McGONAGLE, Dave	General Electric
MOONEY, Charles	Grumman Aerospace
MORSE, H.R.	Frey Federal Systems
OBERNDORF, Tricia	Naval Ocean Systems Center
OLDHAM, John	TRW
REEDY, Ann	Planning Research Corporation
RUBY, Jim	Hughes Aircraft Corporation

SAIB, Sabina	General Research Corporation
SIBLEY, Edgar	Alpha Omega Group, Inc.
STANDISH, Thomas	University of California at Irvine
WESTERMANN, Rob	TNO-IBBC (The Netherlands)
WILLMAN, Hero	Raytheon
WREGE, Doug	Control Data Corporation
YELOWITZ, Larry	Ford Aerospace and Communications
Members unable to attend:	
LAHTINEN, Pekka	Oy Softplan Ab
LOVEMAN, Dave	Massachusetts Computer Associates
PLOEDEREDER, Erhard	IABG (Germany)

APPENDIX C DEFINITIONS

INTEROPERABILITY: Interoperability is the ability of APSEs to exchange data base objects and their relationships in forms usable by tools and user programs without conversion. Interoperability is measured in the degree to which this exchange can be accomplished without conversion.

TRANSPORTABILITY: Transportability of an APSE tool is the ability of the tool to be installed on a different KAPSE; the tool must perform with the same functionality in both APSEs. Transportability is measured in the degree to which this installation can be accomplished without reprogramming. Portability and transferability are commonly used synonyms.

REUSABILITY: Reusability is the ability of a program unit to be employed in the design, documentation, and construction of new programs. Reusability is measured in the degree to which this reuse can be accomplished witho t reprogramming.

HOST: A host is a computer system upon which an APSE resides, executes and supports Ada software development and maintenance. Examples are IBM 360/370, VAX 11/780, CDC 6000/7000, DEC 20 and UNIVAC 1110 with any of their respective operating systems.

TARGET: A target is a computer system upon which Ada programs execute.

Remark: Hosts are, in fact, also targets, in as much as the APSE is written in Ada. A target might not be capable of supporting an APSE. An embedded target is a target which is used in mission critical applications. Examples of embedded target computer systems are AN/AYK-14, AN/UYK-43 and computer systems conforming to MIL-STD-1750A and MIL-STD-1862 (NEBULA).

REHOSTABILITY: Rehostability of an APSE is the ability of the APSE to be installed on a different HOST. Rehostability is measured in the degree to which the installation can be accomplished with needed re-programming localized to the KAPSE. Assessment of rehostability includes any needed changes to non-KAPSE components of the APSE, in addition to the changes to the KAPSE.

RETARGETABILITY: Retargetability is the ability of a target-sensitive APSE tool to accomplish the same function with respect to another target. Retargetability is measured in the degree to which this can be accomplished without modifying the tool. Not all tools will have target specific functions.

APPENDIX D Initial KAPSE Interface Categories

A. DATA BASE (DB) SERVICES

A.1 Exactly what object/file storing/accessing/querying/protecting/cross-referencing capabilities are provided by the KAPSE? What are the interface specifications for tools and users to exercise those functions? What limitations and capacities are imposed? What provisions are there for modifying or extending the given mechanisms? How does the DB support installation ("registration") of new APSE tools?

B. CONFIGURATION MANAGEMENT (CM)

B.1 What CM functions or capabilities are presented by the KAPSE? Is this separable from or integrated into the KAPSE DB? Can the scheme be modified or extended?

C. DEVICE INTERACTIONS

C.1 What set of devices are supported by the KAPSE, and what are the specifications for tools and users to interface to them? To what extent does the KAPSE present a generic or parameterizable view of devices?

D. ADA PROGRAM RUN-TIME SYSTEM (RTS) I/O

D.1 What is the specification of the RTS? Are its capabilities usable by users at the command level or by tools directly (other than coding corresponding Ada features)? Includes standard and non-standard I/O supported by the KAPSE.

E. COMMAND LANGUAGE INTERFACES

E.1 Does the KAPSE in any way define or constrain the user command language, or define a set of primitive operators? What KAPSE interfaces are used by a command language processor, and how?

F. SPECIAL-CHARACTER/CONTROL-KEY DEPENDENCIES

F.1 How are non-ASCII, machine-dependent special characters handled by the KAPSE? Are some assigned special meanings at the KAPSE interface level, or are some handled (unprocessed by the KAPSE) by the underlying system? What is the KAPSE assignment of special characters to functions?

G. SUSPENSION/INTERRUPTION/CONTINUATION MECHANISMS

G.1 What capabilities are presented in these areas by the KAPSE? How do tools and users use them? Are there differences in how this works depending whether user is as the top command level versus at a "lower" tool command level?

H. LOGON/LOGOFF SERVICES

H.1 What happens automatically at user logon? What provision exists for "executing" a user "profile" or "mode" file to establish a personally tailored initial set of environment parameters and conditions? For altering that user profile (any automatic changes)? What actions happen automatically at logoff, what options or information is presented to the user, and how does this interact with the logon profile parameters?

I. INTER-TOOL INVOCATION MECHANISMS

I.1 How do APSE tools (and users) invoke other APSE-tools? What are the extents of sequential, concurrent, and suspended interactions between tools? What are the mechanisms for communicating data and control information for each type of invocation?

J. ADA INTERMEDIATE LANGUAGE (IL)

J.1 DIANA is a proposed standard intermediate language to be produced by Ada compiler front-ends. Alternatives may provide different success in achieving varying goals such as target-code optimizability, processing by other tools (e.g., debuggers, analyzers), and host resource conservation.

K. OTHER COMPILER INTERFACES

K.1 Depending on the completeness of the standard IL, compilers may be required to deposit other information deduced from an Ada program into the data base for future usage by static and dynamic analyzers, test-generation and evaluation tools, documentation tools, program provers, and other yet-unforeseen tools.

L. PERFORMANCE MEASUREMENT CAPABILITIES

L.1 What support exists for automatically or manually instrumenting APSE tools and programs to record static and dynamic information on timing, storage, locality, KAPSE and other-tool interactions, device interactions, etc. For evaluating such data, or even tuning KAPSE characteristics based on it?

M. RECOVERY MECHANISMS

M.1 What system failures are anticipated and (partially) handled? What partial recovery is effected, what notification is given to users when this occurs, how is the extent of recovery performed ascertained, and what manual procedures may increase that extent of recovery? In case of unhandled failures, what notification is given and how is it possible to determine the extent and impact of damage.

N. OTHER OPERATING-SYSTEM SERVICES

N.1 For example, access to special clocks and hardware locations, memory control, logical and physical device reconfiguration, access to native mailers and networkers, etc.

O. OTHER TOOL INTERACTION

O.1 Here we investigate the extent and flexibility for new tools to reuse functionality already implemented in other tools. Examples might include KAPSE DB support for individually usable small-grain components out of which MAPSE tools are built, support for gaining knowledge of and viewing and using all externally visible aspects of existing tools, support for communication between concurrently executing tools, and support for tailoring of existing tools.

P. SUPPORT FOR SEPARATE TARGETS

P.1 Is there anything in the KAPSE that provides, promotes, or constrains future tools and capabilities specific to the operational target computer for the software developed using the APSE?

APPENDIX E
GROUP SUMMARIES KAPSE INTERFACE CATEGORIES

GROUP 1. KAPSE User Support

- A. Program Invocation and Control
- B. Logon/Logoff Services
- C. Device Interactions

GROUP 2. Data Interfaces

- D. Database (DB) Services
- E. Inter-Tool Data Interfaces
- F. Ada Intermediate Language (IL)

GROUP 3. KAPSE Services

- G. Ada Program Run-Time System (RTS)
- H. Bindings and Their Effect on Tools
- I. Performance Measurement

GROUP 4. Miscellaneous & 'Non-Categories'

- J. Recovery Mechanisms
- K. Distributed APSE's
- L. Security
- M. Support for Targets
- N. Pragma & Other Tool Controls

KAPSE INTERFACE CATEGORY Title:
(Containing KAPSE INTERFACE GROUP Title:)

1. EXPLANATION (or meaning or definition).

2. KEY ISSUES: this usually portrays the standardization issue in terms of questions whose answers (for proposed KAPSE's) might reveal de facto standard-achieving approaches, divergent approaches, or basic challenges to Stoneman conformance.

3. RELEVANCE to KAPSE standardization & life-cycle cost-saving objectives.

4. PRIORITY (as an abstraction, independent of proposed approaches).

5. RELATED TOPICS (other interface categories). [Note that as a result of changes effected at the January KIT meeting, the current list of interface categories already reflects additions, mergers, and deletions based on an initial analysis of these relations.]

6. PROPOSED APPROACHES:

   6a. ALS
   6b. AIE
   6c. Existing, related standards (e.g., IEEE, CODASYL, ANSI)
   6d. Other proposals

7. Standardization RISK analysis (consequence of failing to achieve standard; probability that current approaches are insufficiently compatible to establish a de facto standard; risk and consequences of over-standardization; cost to ALS and AIE to change design).

8. ACTIONS TO TAKE (by KIT & KITI). At present, at least two separate approaches to achieving needed standardizations are envisioned. For some interface categories, standards for aspects of a KAPSE interface's implementation may be specified; such interface specifications may be the subject of future KAPSE certification procedures. For some categories, life-cycle payoffs will only be achieved by a strategy including APSE-tool-writing standards (conventions and guidelines) in tool-interface areas hardly affected by a KAPSE interface (e.g., the intermediate language); when warranted, the KIT may in the future specify such tool-interface standards and recommend to the AJPO their adoption for DoD tool procurements.

KAPSE INTERFACE WORKSHEET

Date: 17 Feb 82

KAPSE INTERFACE CATEGORY: A. Program Invocation and Control

(KAPSE INTERFACE GROUP: 1. KAPSE User Support)

A1. EXPLANATION:

Program invocation and control includes the generation, transmission and execution of instructions, from a user or from an executing program, with which:

- o Programs may be initiated, suspended, resumed, or terminated;
- o Program parameters and options may be specified;
- o Program execution may be monitored;
- o Program inputs and outputs may be specified or redirected;
- o Program termination status and other messages may be returned.

A2. KEY ISSUES:

- o What facilities are provided for a replaceable command processor tool?
- o What facilities are provided for a program to invoke another program?
- o What facilities are provided for a program to invoke itself?
- o Are any of the program invocation and control facilities specially privileged or constrained?
- o What facilities are provided for a user to determine the status of a running program?
- o What facilities are provided for a user to control a running program? In particular, can a user suspend, resume, or terminate execution?
- o What facilities are provided for a user to direct or redirect program input and output?
- o What facilities are provided for simultaneous execution of several programs for one user?
- o What facilities are provided for creation and use of stored command language scripts?

Date: 17 Feb 82

- What facilities are provided for collection and display of program- or system-generated messages during program execution?

- Other considerations (Group 4 cross-category issues):

  - Do proposed approaches limit the capability of the KAPSE, or APSE tools, to effect APSE system recovery? [see category J.]

  - Do proposed approaches limit extension of APSE's to distributed systems? [see category K.]

  - Do proposed approaches limit design and implementation in support of future system Security requirements? [see category L.]

  - Do proposed approaches limit extension of APSE capabilities for Target-Dependent functionality? [see category M.]

A3. RELEVANCE:

A4. PRIORITY:

A5. RELATED CATEGORIES:

A6. PROPOSED APPROACHES:

6a. ALS:

6b. AIE:

6c. Existing Standards:

6d. Other:

A7. RISK:

A8. ACTIONS TO TAKE:

[Preliminary formulation of long-term KIT approach:]

- Establish interface specifications to be effected by a KAPSE, potentially a subject of future KAPSE certification procedures.

Date: 17-Feb-82 (Form Rev. 8-Feb-82)

KAPSE INTERFACE WORKSHEET

Date: 17 Feb 82

KAPSE INTERFACE CATEGORY: B. Logon/Logoff Services

(KAPSE INTERFACE GROUP: 1. KAPSE User Support)

B1. EXPLANATION:

Logon/logoff services generally include granting system access to a user based on access codes and accounting verification, establishing a personalized working environment for a user, connecting a user to a collection of database objects, accounting for resources used, collecting system-generated messages, disconnecting the user and saving his working environment when requested, and disposing of programs and database objects left behind after user logoff. Many of these services implement installation-unique or project-unique operational policies and procedures.

B2. KEY ISSUES:

- o What standard logon/logoff facilities are provided?
- o What is the interface between logon/logoff with the host system and logon/logoff with the APSE?
- o What facilities are provided for installation and invocation of tools that provide user logon/logoff services?
- o Other considerations (Group 4 cross-category issues):
  - Do proposed approaches limit the capability of the KAPSE, or APSE tools, to effect APSE system recovery? [see category J.]
  - Do proposed approaches limit extension of APSE's to distributed systems? [see category K.]
  - Do proposed approaches limit design and implementation in support of future system Security requirements? [see category L.]

B3. RELEVANCE:

B4. PRIORITY:

B5. RELATED CATEGORIES:

B6. PROPOSED APPROACHES:

6a. ALS:

6b. AIE:

Date: 17 Feb 82

6c. Existing Standards:

6d. Other:

B7. RISK:

B8. ACTIONS TO TAKE:

[Preliminary formulation of long-term KIT approach:]

- Establish interface specifications to be effected by a KAPSE, potentially a subject of future KAPSE certification procedures.

Date: 17-Feb-82 (Form Rev. 8-Feb-82)

K A P S E I N T E R F A C E W O R K S H E E T

Date: 17 Feb 82

KAPSE INTERFACE CATEGORY: C. Device Interactions

(KAPSE INTERFACE GROUP: 1. KAPSE User Support)

C1. EXPLANATION:

Device interactions include transmission of data and control information between programs and all forms of input/output devices. This category also covers character sets, communication protocols, device service routines in the KAPSE and host system, and high-level and low-level input/output interfaces in the Ada language.

C2. KEY ISSUES:

- o What set of devices is supported?
- o What virtual device interfaces are provided?
- o What device specific interfaces are provided?
- o What are the existing interface specifications?
- o What generic or parameterized device facilities are provided?
- o What facilities are provided for new kinds of I/O devices?
- o What facilities are provided for substitution of a process for a device, or vice versa?
- o What facilities are provided for interruption and resumption of the flow of data between processes and devices?
- o What facilities are provided for recovery after accidental disconnection of devices?
- o Can all 256 8-bit character codes be transmitted between devices and processes? If not, what is the disposition of characters that are not transmitted?
- o Other considerations (Group 4 cross-category issues):
  - Do proposed approaches limit the capability of the KAPSE, or APSE tools, to effect APSE system recovery? [see category J.]
  - Do proposed approaches limit extension of APSE's to distributed systems? [see category K.]
  - Do proposed approaches limit design and implementation in

support of future system Security requirements? [see category L.]

- Do proposed approaches limit extension of APSE capabilities for Target-Dependent functionality? [see category M.]

C3. RELEVANCE:

C4. PRIORITY:

C5. RELATED CATEGORIES:

D. (Data Base Services)

G(?) (Ada RTS)

C6. PROPOSED APPROACHES:

6a. ALS:

6b. AIE:

6c. Existing Standards:

6d. Other:

C7. RISK:

C8. ACTIONS TO TAKE:

[Preliminary formulation of long-term KIT approach:]

- Establish interface specifications to be effected by a KAPSE, potentially a subject of future KAPSE certification procedures.

- Develop proposed APSE-tool-writing interface standards in this area (e.g., for device drivers).

Date: 17-Feb-82 (Form Rev. 8-Feb-82)

KAPSE INTERFACE WORKSHEET

Date: 17 Feb 82

KAPSE INTERFACE CATEGORY: D. Database (DB) Services

(KAPSE INTERFACE GROUP: 2. Data Interfaces)

D1. EXPLANATION:

The KAPSE Database Services comprise those facilities made available to KAPSE-level tools in the following subareas:

a) Basic Object Operations: create, delete, open, close, etc.

b) Attribute Manipulation and Query: create attribute, delete attribute, modify attribute, list objects with specified attributes, etc.

c) Categories: definition and semantics of standard categories, provision for user-defined categories

d) Abstract Objects and Version Control: object, version, and revision naming conventions, functions for version and revision creation, access, and manipulation

e) Access Control: syntax and semantics of the 'access' attribute

f) History Maintenance: functions that maintain and access the 'history' attribute

g) Partitions: semantics of, and functions that operate on partitions; provision for a standard, initial partition structure

h) Archive and Back-up

i) Configuration Management: support for the configuration management function (recognizing that this function will likely be supplied by a KAPSE-level tool)

D2. KEY ISSUES:

- Many of these interfaces potentially have deep implications on the structure of the underlying database. It will be a considerable challenge to produce a useful standard that does not overly constrain the implementation. Parts of both Ada and Stoneman have been criticized for precluding efficient implementations. In our standardization efforts we must very carefully weigh power and flexibility v. simplicity and efficiency.

- Within the Database Services, these considerations have

particular impact on the design of interfaces for query, version control, history maintenance, and partitions.

- o The other key issue is the factorization of functionality between the KAPSE and MAPSE, especially with regards to configuration management. The issue also surfaces with regard to the command language interpreter, in defining a partition search path for looking up the names of objects.

- o A significant overlap that needs to be resolved is that of input/output. These are now in a separate category (device interactions). It should be decided fairly soon whether devices, at the MAPSE-level, are to be treated as database objects.

- o Other considerations (Group 4 cross-category issues):

  - Do proposed approaches limit the capability of the KAPSE, or APSE tools, to effect APSE system recovery? [see category J.]

  - Do proposed approaches limit extension of APSE's to distributed systems? [see category K.]

  - Do proposed approaches limit design and implementation in support of future system Security requirements? [see category L.]

  - Do proposed approaches limit extension of APSE capabilities for Target-Dependent functionality? [see category M.]

D3. RELEVANCE:

D4. PRIORITY:

D5. RELATED CATEGORIES:

D6. PROPOSED APPROACHES:

6a. ALS:

6b. AIE:

6c. Existing Standards:

6d. Other:

D7. RISK:

D8. ACTIONS TO TAKE:

[Preliminary formulation of long-term KIT approach:]

- Establish interface specifications to be effected by a

Date: 17 Feb 82

KAPSE, potentially a subject of future KAPSE certification procedures.

Date: 17-Feb-82 (Form Rev. 8-Feb-82)

Date: 17 Feb 82

KAPSE INTERFACE CATEGORY: E. Inter-Tool Data Interfaces

(KAPSE INTERFACE GROUP: 2. Data Interfaces)

E1. EXPLANATION:

These are the non-IL data interfaces for all MAPSE-level tools. In addition to the tools listed under the Ada Intermediate Language Category, the following tools must be considered:

a) Documentation tools
b) Requirements and specification tools
c) Project management tools
d) Configuration management tools

Standards and conventions for the formats of all data to be transmitted between or shared by these tools are to be investigated.

E2. KEY ISSUES:

- Two data interfaces are of immediate interest: standard program library format standard object file format. Again, standardization must take care not to overly constrain the implementation of tools that access these formats, and must allow these tools to be implemented efficiently.

- A major issue with respect to the other tool areas listed is whether this sort of standardization properly falls within the purview of the KIT, and if so, whether this standardization (particularly in the areas of requirements, specification, and project management) is feasible in the expected lifetime of the KIT.

- Other considerations (Group 4 cross-category issues):

  - Do proposed approaches limit the capability of the KAPSE, or APSE tools, to effect APSE system recovery? [see category J.]

  - Do proposed approaches limit extension of APSE's to distributed systems? [see category K.]

  - Do proposed approaches limit design and implementation in support of future system Security requirements? [see category L.]

  - Do proposed approaches limit extension of APSE capabilities for Target-Dependent functionality? [see category M.]

E3. RELEVANCE:

KAPSE INTERFACE WORKSHEET

Date: 17 Feb 82

E4. PRIORITY:

E5. RELATED CATEGORIES:

C. (Device Interactions)

E6. PROPOSED APPROACHES:

6a. ALS:

6b. AIE:

6c. Existing Standards:

6d. Other:

E7. RISK:

E8. ACTIONS TO TAKE:

[Preliminary formulation of long-term KIT approach:]

- Develop proposed APSE-tool-writing interface standards in this area (e.g., common file formats).

Date: 17-Feb-82 (Form Rev. 8-Feb-82)

KAPSE INTERFACE WORKSHEET

Date: 17 Feb 82

KAPSE INTERFACE CATEGORY: F. Ada Intermediate Language (IL)

(KAPSE INTERFACE GROUP: 2. Data Interfaces)

F1. EXPLANATION:

The Ada Intermediate Language must serve the primary function of retaining Ada program information for:

a) separate compiler phases
b) optimizers
c) static and dynamic analyzers
d) debuggers
e) test tools
f) pretty-printers
g) syntax-oriented editors

F2. KEY ISSUES:

- It appears likely that the main involvement of the KIT in IL issues will be in the area of requirements and policy. Tartan has been contracted to firm up the details of DIANA, and we should work closely with them.

- Immediate issues are those of extensibility and standardization of representation. We must be careful to set a standard that will promote true portability, not just portability in the abstract.

- Other consideration (Group 4 cross-category issues):

  - Do proposed approaches limit extension of APSE capabilities for Target-Dependent functionality? [see category M.]

F3. RELEVANCE:

F4. PRIORITY:

F5. RELATED CATEGORIES:

F6. PROPOSED APPROACHES:

6a. ALS:

6b. AIE:

6c. Existing Standards:

6d. Other:

F7. RISK:

KAPSE INTERFACE WORKSHEET

Date: 17 Feb 82

F8. ACTIONS TO TAKE:

[Preliminary formulation of long-term KIT approach:]

- Develop proposed APSE-tool-writing interface standards in this area.

Date: 17-Feb-82 (Form Rev. 8-Feb-82)

KAPSE INTERFACE WORKSHEET

Date: 17 Feb 82

KAPSE INTERFACE CATEGORY: G. Ada Program Run-Time System (RTS)

(KAPSE INTERFACE GROUP: 3. KAPSE Services)

G1. EXPLANATION:

The Ada RTS provides the basic run-time support facilities that are required by Ada programs that execute within the APSE. This may include, but is not limited to, closed-routine (non-inline code) provision for multitasking, exception handling, certain arithmetic operations, certain string operations, program initialization/termination, and standard package INPUT_OUTPUT capabilities. Inclusion of needed RTS routines is often envisioned as a link-time input (from a standard library of linkables); for linkage of APSE tools, that may not be the case as the KAPSE may provide a standard interface to 'resident' components providing all Ada RTS capabilities.

G2. KEY ISSUES:

- o What is the specification of the run-time support system?
- o What privileges should be specifiable, grantable, delegatable, or revocable with respect to RTS accessibility as a function of user, project, module, process, etc. at the command language level? By tools directly?
- o What should be the relationship between command-language input form of expression and tool interface to KAPSE?
- o Other considerations (Group 4 cross-category issues):
  - Do proposed approaches limit extension of APSE's to distributed systems? [see category K.]
  - Do proposed approaches limit design and implementation in support of future system Security requirements? [see category L.]

G3. RELEVANCE:

G4. PRIORITY:

G5. RELATED CATEGORIES:

C(?) (Device Interactions)

G6. PROPOSED APPROACHES:

6a. ALS:

KAPSE INTERFACE WORKSHEET

Date: 17 Feb 82

6b. AIE:

6c. Existing Standards:

6d. Other:

G7. RISK:

G8. ACTIONS TO TAKE:

[Preliminary formulation of long-term KIT approach:]

- Establish interface specifications to be effected by a KAPSE; potentially a subject of future KAPSE certification procedures.

- Or, consider possibility that this category may be removed from KIT purview in that ACVC certification of an Ada compiler may establish sufficient RTS standardization for the APSE which is the environment of the compiler's validation (and yielding tool portability (identical function) when a tool is recompiled by another certified Ada compiler in another APSE).

Date: 17-Feb-82 (Form Rev. 8-Feb-82)

Date: 17 Feb 82

KAPSE INTERFACE CATEGORY: H. Bindings and Their Effect on Tools

(KAPSE INTERFACE GROUP: 3. KAPSE Services)

H1. EXPLANATION:

Binding is the assigning of a value or referent to an identifier. Here we are concerned with link- or execution-time assignment of concrete subprograms, devices, and data-base objects to identifiers in Ada programs.

H2. KEY ISSUES:

- o What operations must be supported by the KAPSE regarding diagnostic, interpretive, and full-speed execution of programs incorporating static and dynamic binding to subprograms and to data?
- o What are facilities for making and exploiting a commitment to static binding?
- o What facilities exist for deferring commitment with respect to binding and exploiting the deferral of commitment for debugging, exploration, evaluation, withdrawal, and change of commitment?
- o What facility should exist for measuring cost/benefits of alternative forms of binding?
- o Other considerations (Group 4 cross-category issues):
  - Do proposed approaches limit the capability of the KAPSE, or APSE tools, to effect APSE system recovery? [see category J.]
  - Do proposed approaches limit extension of APSE's to distributed systems? [see category K.]
  - Do proposed approaches limit design and implementation in support of future system Security requirements? [see category L.]
  - Do proposed approaches limit extension of APSE capabilities for Target-Dependent functionality? [see category M.]

H3. RELEVANCE:

H4. PRIORITY:

H5. RELATED CATEGORIES:

Date: 17 Feb 82

H6. PROPOSED APPROACHES:

6a. ALS:

6b. AIE:

6c. Existing Standards:

6d. Other:

H7. RISK:

H8. ACTIONS TO TAKE:

[Preliminary formulation of long-term KIT approach:]

- Establish interface specifications to be effected by a KAPSE, potentially a subject of future KAPSE certification procedures.

Date: 17-Feb-82 (Form Rev. 8-Feb-82)

KAPSE INTERFACE WORKSHEET

Date: 17 Feb 82

KAPSE INTERFACE CATEGORY: I. Performance Measurement

(KAPSE INTERFACE GROUP: 3. KAPSE Services)

I1. EXPLANATION:

Performance measurement is the selection, collection, and recording of dynamic information on APSE activity and resource utilization as a function of time. From this follows performance evaluation, which is the technical assessment of a system (APSE) or system component (APSE tool or KAPSE component) to determine how effectively operating objectives have been achieved.

I2. KEY ISSUES:

- o What KAPSE support mechanisms should exist to support selection, collection and recording of dynamic information on process resource utilization as a function of time? What KAPSE support mechanisms exist for tuning KAPSE characteristics based on evaluation of such information?
- o What access should exist to special clocks and hardware locations, memory control, logical and physical device reconfiguration, native mailers and networks, etc.?
- o What mechanisms should exist to support high-level event recording? Examples of high-level events are:
  - Login/Logoff
  - File Open/Close
  - History of Tool invocation
  - Trace Information
- o Other considerations (Group 4 cross-category issues):
  - Do proposed approaches limit the capability of the KAPSE, or APSE tools, to effect APSE system recovery? [see category J.]
  - Do proposed approaches limit extension of APSE's to distributed systems? [see category K.]
  - Do proposed approaches limit design and implementation in support of future system Security requirements? [see category L.]
  - Do proposed approaches limit extension of APSE

capabilities for Target-Dependent functionality? [see category M.]

I3. RELEVANCE:

I4. PRIORITY:

I5. RELATED CATEGORIES:

I6. PROPOSED APPROACHES:

6a. ALS:

6b. AIE:

6c. Existing Standards:

6d. Other:

I7. RISK:

I8. ACTIONS TO TAKE:

[Preliminary formulation of long-term KIT approach:]

- Establish interface specifications to be effected by a KAPSE, potentially a subject of future KAPSE certification procedures.

Date: 17-Feb-82 (Form Rev. 8-Feb-82)

Date: 17 Feb 82

KAPSE INTERFACE CATEGORY: J. Recovery Mechanisms

(KAPSE INTERFACE GROUP: 4. Miscellaneous & 'Non-Categories')

J1. EXPLANATION:

It is a Stoneman guideline that an APSE be a highly robust system that can protect itself from user and system errors, that can recover from unforeseen situations and that can provide meaningful diagnostic information to its users.

J2. KEY ISSUES:

[This proposed category (especially a design approach achieving it) is very intertwined with most of the categories in Groups 1-3. Hence, we postulate that its standardized achievement by a KAPSE is probably a fall-out of achieving effective standardization in those other categories, if Recovery-Mechanism considerations are factored into standardization approaches for those categories. To that end, a relevant question has been added to the list of Key Issues for each of those other categories whose approach is thought to impact Recovery Mechanisms. The worksheet for this category will not be deleted until thorough analysis confirms that achievement of this goal is possible by making Recovery Mechanisms an orthogonal dimension of consideration for most other categories. Following is the question for this category which is factored across the other categories:]

- Do proposed approaches limit the capability of the KAPSE, or APSE tools, to effect APSE system recovery? [see category J.]

J3. RELEVANCE:

J4. PRIORITY:

J5. RELATED CATEGORIES:

A, B, C, D, E, H, I

J6. PROPOSED APPROACHES

6a. ALS:

6b. AIE:

6c. Existing Standards:

6d. Other:

J7. RISK:

KAPSE INTERFACE WORKSHEET

Date: 17 Feb 82

J8. ACTIONS TO TAKE:

- Continue analyses to identify complete incorporation of these considerations into other categories, and to determine when and if this category should be deleted as an independent KIT interface category.

Date: 17-Feb-82 (Form Rev. 8-Feb-82)

K A P S E I N T E R F A C E W O R K S H E E T

Date: 17 Feb 82

KAPSE INTERFACE CATEGORY: K. Distributed APSE's

(KAPSE INTERFACE GROUP: 4. Miscellaneous & "Non-Categories")

K1. EXPLANATION:

An APSE is distributed if its KAPSE, tools, or data base reside on or execute on a configuration including more than one host computer system.

K2. KEY ISSUES:

[This proposed category (especially a design approach achieving it) is very intertwined with most of the categories in Groups 1-3. Hence, we postulate that its standardized achievement by a KAPSE is probably a fall-out of achieving effective standardization in those other categories, if APSE-Distribution considerations are factored into standardization approaches for those categories. To that end, a relevant question has been added to the list of Key Issues for each of those other categories whose approach is thought to impact APSE Distribution. The worksheet for this category will not be deleted until thorough analysis confirms that achievement of this goal is possible by making APSE Distribution an orthogonal dimension of consideration for most other categories. Following is the question for this category which is factored across the other categories:]

- Do proposed approaches limit extension of APSE's to distributed systems? [see category K.]

K3. RELEVANCE:

K4. PRIORITY:

K5. RELATED CATEGORIES:

A, B, C, D, E, G, H, I

K6. PROPOSED APPROACHES:

6a. ALS:

6b. AIE:

6c. Existing Standards:

6d. Other:

K7. RISK:

K8. ACTIONS TO TAKE:

KAPSE INTERFACE WORKSHEET

Date: 17 Feb 82

- Continue analyses to identify complete incorporation of these considerations into other categories, and to determine when and if this category should be deleted as an independent KIT interface category.

Date: 17-Feb-82 (Form Rev. 8-Feb-82)

K A P S E I N T E R F A C E W O R K S H E E T

Date: 17 Feb 82

KAPSE INTERFACE CATEGORY: L. Security

(KAPSE INTERFACE GROUP: 4. Miscellaneous & 'Non-Categories')

L1. EXPLANATION:

Computer security includes issues of proper isolation of separate users (and their APSE objects) from each other, and issues of multi-level DoD security classification schemes when an APSE is to contain and handle classified data and tools.

L2. KEY ISSUES:

[This proposed category (especially a design approach achieving it), including multi-level DoD classification schemes, is very intertwined with most of the categories in Groups 1-3. Hence, we postulate that its standardized achievement by a KAPSE is probably a fall-out of achieving effective standardization in those other categories, if Security considerations are factored into standardization approaches for those categories. To that end, a relevant question has been added to the list of Key Issues for each of those other categories whose approach is thought to impact Security. The worksheet for this category will not be deleted until thorough analysis confirms that achievement of this goal is possible by making Security an orthogonal dimension of consideration for most other categories. Following is the question for this category which is factored across the other categories:]

- Do proposed approaches limit design and implementation in support of future system Security requirements? [see category L.]

L3. RELEVANCE:

L4. PRIORITY:

L5. RELATED CATEGORIES:

A, B, C, D, E, G, H, I

L6. PROPOSED APPROACHES:

6a. ALS:

6b. AIE:

6c. Existing Standards:

6d. Other:

L7. RISK:

NAPSE INTERFACE WORKSHEET

Date: 17 Feb 82

L8. ACTIONS TO TAKE:

- Continue analyses to identify complete incorporation of these considerations into other categories, and to determine when and if this category should be deleted as an independent KIT interface category.

Date: 17-Feb-82 (Form Rev. 8-Feb-82)

K A P S E I N T E R F A C E W O R K S - E E T

Date: 17 Feb 82

KAPSE INTERFACE CATEGORY: M. Support for Targets

(KAPSE INTERFACE GROUP: 4. Miscellaneous & 'Non-Categories')

M1. EXPLANATION:

Support for Targets means tools and capabilities specific to the operational target computer, not the APSE host. This includes loading and executing, debugging, testing, evaluation, and maybe even maintenance in host-target configurations of varying modes, including (but not limited to) target simulators (as APSE tools), physical environment simulations (APSE tools or connected microprocessors), coupled target emulations or in-circuit emulations, and/or physical instances of the target processors (at varying degrees of coupling) connected to the APSE host.

M2. KEY ISSUES:

[This proposed category (especially a design approach achieving it) is very intertwined with most of the categories in Groups 1-3. Hence, we postulate that its standardized achievement by a KAPSE is at least partially a fall-out of achieving effective standardization in those other categories, if considerations for Target-dependent extensions of functionality are factored into standardization approaches for those categories. To that end, a relevant question has been added to the list of Key Issues for each of those other categories whose approach is thought to impact Target Support. Following is the question factored across other categories:]

- Do proposed approaches limit extension of APSE capabilities for Target-Dependent functionality? [see category M.]

[Because of severe criticisms of both the ALS and AIE designs on this issue from a broad segment of the ECS contractor community, it is presumed that this category will also be preserved as an independent evaluation viewpoint, and that in the future issues and approaches may be identified germane primarily to this category. In no event will the worksheet for this category be deleted until thorough analysis confirms that achievement of this goal is possible by making Target Support an orthogonal dimension of consideration for most other categories.]

M3. RELEVANCE:

M4. PRIORITY:

KAPSE INTERFACE WORKSHEET

Date: 17 Feb 82

M5. RELATED CATEGORIES:

A, C, D, E, F, H, I

M6. PROPOSED APPROACHES:

6a. ALS:

6b. AIE:

6c. Existing Standards:

6d. Other:

M7. RISK:

M8. ACTIONS TO TAKE:

- Continue analyses to identify complete incorporation of these considerations into other categories, and to determine when and if this category should be deleted as an independent KIT interface category. If not to be deleted, identify capabilities and issues distinguishing target support from host support.

Date: 17-Feb-82 (Form Rev. 8-Feb-82)

K A P S E I N T E R F A C E W O R K S H E E T

Date: 17 Feb 82

KAPSE INTERFACE CATEGORY: N. Prasma & Other Tool Controls

(KAPSE INTERFACE GROUP: 4. Miscellaneous & 'Non-Categories')

N1. EXPLANATION:

This concerns the need to standardize Prasmas (and other special 'non-Ada' capabilities) that the implementation of APSE tools may employ, and hence the need for standardized compiler specification (potentially under the purview of the ACVC) of a selected set of Prasmas and other tool controls, in order to achieve APSE tool portability.

N2. KEY ISSUES:

[New issue; identified but not analyzed at KIT meeting.]

N3. RELEVANCE:

N4. PRIORITY:

N5. RELATED CATEGORIES:

N6. PROPOSED APPROACHES:

6a. ALS:

6b. AIE:

6c. Existing Standards:

6d. Other:

N7. RISK:

N8. ACTIONS TO TAKE:

- Discuss potential problem; identify likely uses of non-standard Prasmas and other capabilities by APSE tools and assess their impact on portability; determine if threat to KIT objectives exists and, if so, retain this category and identify appropriate actions.

Date: 17-Feb-82 (Form Rev. 8-Feb-82)


ILMED
8